Lecture Notes in Computer Science 3603

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Joe Hurd Tom Melham (Eds.)

Theorem Proving in Higher Order Logics

18th International Conference, TPHOLs 2005
Oxford, UK, August 22-25, 2005
Proceedings

Springer

Volume Editors

Joe Hurd
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
E-mail: joe.hurd@comlab.ox.ac.uk

Tom Melham
Oxford University Computing Laboratory
Wolfson Building, Parks Road Oxford, OX1 3QD, UK
E-mail: Tom.Melham@comlab.ox.ac.uk

Library of Congress Control Number: 2005930490

CR Subject Classification (1998): F.4.1, I.2.3, F.3.1, D.2.4, B.6.3

ISSN 0302-9743
ISBN-10 3-540-28372-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28372-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11541868 06/3142 5 4 3 2 1 0

Preface

This volume constitutes the proceedings of the *18th International Conference on Theorem Proving in Higher Order Logics* (TPHOLs 2005), which was held during 22–25 August 2005 in Oxford, UK. TPHOLs covers all aspects of theorem proving in higher order logics as well as related topics in theorem proving and verification.

There were 49 papers submitted to TPHOLs 2005 in the full research category, each of which was refereed by at least three reviewers selected by the program committee. Of these submissions, 20 research papers and 4 proof pearls were accepted for presentation at the conference and publication in this volume. In keeping with longstanding tradition, TPHOLs 2005 also offered a venue for the presentation of work in progress, where researchers invited discussion by means of a brief introductory talk and then discussed their work at a poster session. A supplementary proceedings volume was published as a 2005 technical report of the Oxford University Computing Laboratory.

The organizers are grateful to Wolfgang Paul and Andrew Pitts for agreeing to give invited talks at TPHOLs 2005.

The TPHOLs conference traditionally changes continents each year to maximize the chances that researchers from around the world can attend. Starting in 1993, the proceedings of TPHOLs and its predecessor workshops have been published in the Springer Lecture Notes in Computer Science series:

1993 (Canada)	Vol. 780	2000 (USA)	Vol. 1869
1994 (Malta)	Vol. 859	2001 (UK)	Vol. 2152
1995 (USA)	Vol. 971	2002 (USA)	Vol. 2410
1996 (Finland)	Vol. 1125	2003 (Italy)	Vol. 2758
1997 (USA)	Vol. 1275	2004 (USA)	Vol. 3223
1998 (Australia)	Vol. 1479	2005 (UK)	Vol. 3603
1999 (France)	Vol. 1690		

We would like to thank our local organizers Ed Smith and Ashish Darbari for their help in many aspects of planning and running TPHOLs.

Finally, we thank our sponsors: Intel Corporation and the EPSRC UK Network in Computer Algebra.

June 2005

Joe Hurd and Tom Melham
TPHOLs 2005 Chairs

Organization

Program Committee

Mark Aagaard (Waterloo)
David Basin (ETH Zürich)
Ching-Tsun Chou (Intel)
Amy Felty (Ottawa)
Jacques Fleuriot (Edinburgh)
Elsa Gunter (UIUC)
Jason Hickey (Caltech)
Joe Hurd (Oxford)
Thomas Kropf (Tübingen & Bosch)
John Matthews (Galois)
Tobias Nipkow (München)
Christine Paulin-Mohring (Paris Sud)
Frank Pfenning (CMU)
Sofiène Tahar (Concordia)
Clark Barrett (NYU)
Yves Bertot (INRIA)
Thierry Coquand (Chalmers)
Jean-Christophe Filliâtre (Paris Sud)
Jim Grundy (Intel)
John Harrison (Intel)
Peter Homeier (US DoD)
Paul Jackson (Edinburgh)
Pete Manolios (Georgia Tech)
César Muñoz (Nat. Inst. Aerospace)
Sam Owre (SRI)
Lawrence Paulson (Cambridge)
Konrad Slind (Utah)
Burkhart Wolff (ETH Zürich)

Additional Referees

Amr Abdel-Hamid
Behzad Akbarpour
Tamarah Arons
Sylvain Conchon
Pierre Corbineau
Peter Dillinger
Lucas Dixon
Guillaume Dufay
Marcio Gemeiro
Ganesh Gopalakrishnan
Ali Habibi
Hugo Herbelin
Doug Howe
Robert Jones
Roope Kaivola
Felix Klaedtke
Farhad Mehta
Laura Meikle
Julien Narboux
Lee Pike
Tom Ridge
Hassen Saidi
Christelle Scharff
N. Shankar
Radu Siminiceanu
Sudarshan Srinivasan
Ashish Tiwari
Daron Vroon

Table of Contents

Invited Papers

Regular Papers

Proof Pearls

On the Correctness of Operating System Kernels

Mauro Gargano*, Mark Hillebrand*, Dirk Leinenbach*,**, and Wolfgang Paul

Saarland University, Computer Science Dept., 66123 Saarbrücken, Germany
{gargano, mah, dirkl, wjp}@wjpserver.cs.uni-sb.de

Abstract. The Verisoft project aims at the pervasive formal verification of entire computer systems. In particular, the seamless verification of the *academic system* is attempted. This system consists of hardware (processor and devices) on top of which runs a microkernel, an operating system, and applications. In this paper we define the computation model CVM (communicating virtual machines) in which concurrent user processes interact with a generic microkernel written in C. We outline the correctness proof for concrete kernels, which implement this model. This result represents a crucial step towards the verification of a kernel, e.g. that in the academic system. We report on the current status of the formal verification.

1 Introduction

There is no need to argue about the importance of computer security [1] and operating system security is in the center of computer security. Making operating systems comfortable and at the same time utmost reliable is extremely hard. However, some small and highly reliable operating system kernels, e.g. [2,3,4], have been developed. A reliable kernel opens the way to uncouple the safety-critical applications running under an operating system from the non-critical ones. One runs *two* operating systems under a trusted kernel, a small trusted one for the safety-critical applications and a conventional one for all others. This minimizes the total size of the trusted components. For example, [5] describes a small operating system and Linux running under the L4 microkernel [6].

For critical applications one wishes of course to estimate, how much trust one should put into a system. For this purpose the *common criteria* for information technology security evaluation [7] define a hierarchy of *evaluation assurance levels* EAL-1 to EAL-7. These are disciplines for reviewing, testing / verifying, and documenting systems during and after development. Even the highest assurance level, EAL-7, does not require formal verification of the system implementation. Clearly, the common criteria, in the current revision, stay *behind* the state of the art available at that time: already nine years before Bevier [8] reported on the full formal verification of KIT, a small multitasking operating system kernel written in machine language. KIT implements a fixed number of processes, each occupying a fixed portion of the processor's memory. It provides the following verified services: process scheduling, error handling, message passing, and an interface to asynchronous devices. In terms of complexity, KIT is near to small real-time operating systems like e.g. OSEKTime [9].

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

** Work supported by DFG Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 1–16, 2005.

In this paper we outline an approach to the pervasive verification of a considerably more powerful kernel, supporting virtual memory, memory management, system calls, user defined interrupts, etc. We outline substantial parts of its correctness proof. We report on the current status of the formal verification. The results presented in this paper were obtained in and are of crucial importance to the Verisoft project [10], funded by the German Federal Government. Verisoft has the mission to provide the technology for the formal pervasive verification of entire computer systems of industrial complexity.

2 Overview

To handle the design complexity, computer systems are organized in layers some of which are modeled by well established formal models. Examples are (i) the hardware layer that is modeled by switching circuits and memory components, (ii) the machine language layer that is modeled by random access machines [11] with an appropriate instruction set, and (iii) the programming language layer, e.g. for C, is, for operational semantics, modeled by abstract interpreters, also called abstract C machines. Correctness theorems for components of computer systems are often simulation theorems between *adjacent* layers. Processor correctness concerns a simulation between Layers (i) and (ii). Compiler correctness concerns a simulation between Layers (ii) and (iii).

Aiming at formulating and proving a correctness theorem for an operating system kernel we take a similar approach. We introduce an abstract parallel model of computation called *communicating virtual machines* (CVM) that formalizes concurrent user processes interacting with an operating system kernel. In this model user processes are virtual machines, i.e. processors with virtual memory. The so-called *abstract kernel* is represented as an abstract C machine. Beyond the usual C functions the abstract kernel can call a few special functions, called the *CVM primitives*, that alter the configuration of user processes. For instance, there are CVM primitives to increase / decrease the memory size of a user process or to copy data between user processes (and I/O devices).

By linking abstract kernels with a program implementing the CVM functionality we obtain the *concrete kernel*. In particular, the concrete kernel contains the implementation of the CVM primitives and the implementation of handlers for page faults (not visible in the abstract model). A crucial observation is that the concrete kernel *necessarily* contains assembler code because neither processor registers nor user processes are visible in the variables of a C program. Thus the correctness theorem for the concrete kernel will establish a simulation between CVM and Layer (ii) instead of Layer (iii). Since reasoning on assembler level is tedious we minimize its use in the concrete kernel.

The remainder of this paper is structured as follows. In Sect. 3 we define virtual machines and summarize results from [12] on the simulation of virtual machines by physical machines, processors with physical and swap memory. In Sect. 4 we define abstract $C0$ machines and summarize the compiler correctness proof from [13]. In Sect. 5 we define the CVM model using virtual machines to model computation of the user and abstract $C0$ machines to model computation of an abstract kernel. Section 6 sketches the construction of the concrete kernel containing the CVM implementation. We state the correctness proof for the concrete kernel and outline its proof. In Sect. 7 we report on the status of the formal verification. In Sect. 8 we conclude and sketch further work.

3 Virtual Memory Simulation

Let us introduce some notation. We denote bitvectors by $a \in \{0,1\}^n$. Bit j of bitvector a is denoted by $a[j]$, the sub bitvector consisting of bits j to k (with $k < j$) is denoted by $a[j:k]$. The concatenation of two bitvectors $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ is denoted by $a \circ b \in \{0,1\}^{n+m}$. Occasionally we will abuse notation and identify bitvectors a with their value $\langle a \rangle = \sum_i a[i] \cdot 2^i$ and vice versa. Arithmetic is modulo 2^n. We model memories m as mappings from addresses $a \in \{0,1\}^{32}$ to byte values $m(a) \in \{0,1\}^8$. For natural numbers d we denote by $m_d(a)$ the content of d consecutive memory cells starting at address a, so $m_d(a) = m(a+d-1) \circ \cdots \circ m(a)$.

In the following sub sections we summarize results from [12].

3.1 Virtual Machines

Virtual machines consist of a processor operating on a (uniform) virtual memory. Configurations c_{V} of virtual machines have the following components:

- $c_{\mathrm{V}}.R \in \{0,1\}^{32}$ for a variety of processor registers R. We consider here pipelined DLX machines [14] with a delayed branch mechanism that is implemented by *two* program counters, called delayed program counter $c_{\mathrm{V}}.DPC \in \{0,1\}^{32}$ and program counter $c_{\mathrm{V}}.PC \in \{0,1\}^{32}$. For details see [15].
- The size $c_{\mathrm{V}}.V$ of the virtual memory measured in pages of 4K bytes. It defines the set of accessible virtual addresses $VA(c_{\mathrm{V}}) = \{a \in \{0,1\}^{32} \mid a < c_{\mathrm{V}}.V \cdot 4\mathrm{K}\}$. We split virtual addresses $va = va[31:0]$ into page index $va.px = va[31:12]$ and byte index $va.bx = va[11:0]$.
- A byte addressable virtual memory $c_{\mathrm{V}}.vm : VA(c_{\mathrm{V}}) \to \{0,1\}^8$.
- A write protection function $c_{\mathrm{V}}.p : VA(c_{\mathrm{V}}) \to \{0,1\}$ that only depends on the page index of virtual addresses. A virtual address va is write protected if $c_{\mathrm{V}}.p(va) = 1$.

Computation of the virtual machine is modeled by the function δ_{V} that computes for a given configuration c_{V} its successor configuration c'_{V}. The virtual machine accesses the memory in the following situations: it reads the memory to fetch instructions and to execute load instructions, it writes the memory to execute store instructions.

However, any access to a virtual address $va \notin VA(c_{\mathrm{V}})$ or a write access to va with $c_{\mathrm{V}}.p(va) = 1$ is illegal and leads to an exception. For the CVM model (cf. Sect. 5) we do not consider write protected pages and assume $c_{\mathrm{V}}.p(va) = 0$ for all $va \in VA(c_{\mathrm{V}})$.

Note that the effects of exceptions are not defined in a virtual machine model alone but in an extended context of a virtual machine running under a certain operating system (kernel). Also, the size of the virtual memory $c_{\mathrm{V}}.V$ cannot be changed by the virtual machine itself. This is described in more detail in Sect. 5.

3.2 Physical Machines and Address Translation

Physical machines consist of a processor operating on physical memory and swap memory. Configurations c_{P} of physical machines have components $c_{\mathrm{P}}.R$ for processor registers R, $c_{\mathrm{P}}.pm$ for the physical memory, and $c_{\mathrm{P}}.sm$ for the swap memory. The physical machine has several special purpose registers not present in virtual machines, e.g. the

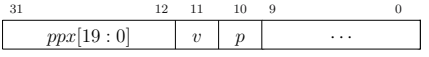


Fig. 1. Page Table Entry

mode register $mode$, the page table origin pto, and the page table length ptl. Computation of the physical machine is modeled by the next state function δ_{P}.

In system mode, i.e. if $c_{\mathrm{P}}.mode = 0$, the physical machine operates almost like a virtual machine with extra registers. In user mode, i.e. $c_{\mathrm{P}}.mode = 1$, memory accesses are subject to address translation: they either cause a page fault or are redirected to the translated physical memory address $pma(c_{\mathrm{P}}, va)$. The result of address translation depends on the contents of the *page table*, a region of the physical memory starting at address $c_{\mathrm{P}}.pto \cdot 4\mathrm{K}$ with $(c_{\mathrm{P}}.ptl + 1)$ entries of four bytes width.

The page table entry address for virtual address va is defined as $ptea(c_{\mathrm{P}}, va) = c_{\mathrm{P}}.pto \cdot 4\mathrm{K} + 4 \cdot va.px$ and the page table entry of va is defined as $pte(c_{\mathrm{P}}, va) = c_{\mathrm{P}}.pm_4(ptea(c_{\mathrm{P}}, va))$. As shown in Fig. 1, a page table entry consists of three components, the physical page index $ppx(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[31 : 12]$, the valid bit $v(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[11]$, and the write protection bit $p(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[10]$.

On user mode memory access to address va, a page fault is signaling if the page index exceeds the page table length, $va.px > c_{\mathrm{P}}.ptl$, if the page table entry is not valid, $v(c_{\mathrm{P}}, va) = 0$, or if for a write access the write protection is active, $p(c_{\mathrm{P}}, va) = 1$. On page fault the page fault handler, an interrupt service, is invoked.

Without a page fault, the access is performed on the (translated) physical memory address $pma(c_{\mathrm{P}}, va)$ defined as the concatenation of the physical page index and the byte index, $pma(c_{\mathrm{P}}, va) = ppx(c_{\mathrm{P}}, va) \circ va.bx$.

For example, the instruction $I(c_{\mathrm{P}})$ fetched in configuration c_{P} is defined as follows. If $c_{\mathrm{P}}.mode = 0$ we define $I(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_4(c_{\mathrm{P}}.DPC)$, otherwise, provided that there is no page fault, we define $I(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_4(pma(c_{\mathrm{P}}, c_{\mathrm{P}}.DPC))$.

3.3 Virtual Memory Simulation

A physical machine with appropriate page fault handlers can simulate virtual machines. For a simple page fault handler, virtual memory is stored on the swap memory of the physical machine and the physical memory acts as a write back cache. In addition to the architecturally defined physical memory address $pma(c_{\mathrm{P}}, va)$, the page fault handler maintains a swap memory address function $sma(c_{\mathrm{P}}, va)$.

We use a simulation relation $B(c_{\mathrm{V}}, c_{\mathrm{P}})$ to indicate that a (user mode) physical machine configuration c_{P} encodes virtual machine configuration c_{V}. Essentially, $B(c_{\mathrm{V}},c_{\mathrm{P}})$ is the conjunction of the following three conditions:

- For every page of virtual memory there is a page table entry in the physical machine, $c_{\mathrm{V}}.V = c_{\mathrm{P}}.ptl + 1$.
- The write protection function of the virtual machine is encoded in the page table, $c_{\mathrm{V}}.p(va) = p(c_{\mathrm{P}}, va)$. As noted earlier in this paper we assume $p(c_{\mathrm{P}}, va) = c_{\mathrm{V}}.p(va) = 0$.
- The virtual memory is stored in physical and swap memory: if $v(c_{\mathrm{P}}, va)$ then $c_{\mathrm{V}}.vm(va) = c_{\mathrm{P}}.pm(pma(c_{\mathrm{P}}, va))$, else $c_{\mathrm{V}}.vm(va) = c_{\mathrm{P}}.sm(sma(c_{\mathrm{P}}, va))$.

The simulation theorem for a single virtual machine has the following form:

Theorem 1. *For all computations* $(c_{\mathrm{V}}^0, c_{\mathrm{V}}^1, \dots)$ *of the virtual machine there is a computation* $(c_{\mathrm{P}}^0, c_{\mathrm{P}}^1, \dots)$ *of the physical machine and there are step numbers* $(s(0), s(1), \dots)$ *such that for all* i *and* $S = s(i)$ *we have* $B(c_{\mathrm{V}}^i, c_{\mathrm{P}}^S)$.

Thus step i of the virtual machine is simulated after step $s(i)$ of the physical machine. Even for a simple handlers, the proof is not completely obvious since a single user mode instruction can cause two page faults. To avoid deadlock and guarantee forward progress, the page fault handler must not swap out the page that was swapped in during the last execution of the page fault handler.

3.4 Synchronization Conditions

If the hardware implementation of a physical machine is pipelined, then an instruction $I(c_{\mathrm{P}}^i)$ that is in the memory stage may modify / affect a later instruction $I(c_{\mathrm{P}}^j)$ for $j > i$ after it has been fetched. It may (i) overwrite the instruction itself, (ii) overwrite its page table entry, or (iii) change the mode. In such situations instruction fetch (in particular translated fetch implemented by a memory management unit) would not work correctly. Of course it is possible to detect such data dependencies in hardware and to roll back the computation if necessary. Alternatively, the software to be run on the processor must adhere to certain *software synchronization conventions*. Let $iaddr(c_{\mathrm{P}}^j)$ denote the address of instruction $I(c_{\mathrm{P}}^j)$, possibly translated. If $I(c_{\mathrm{P}}^i)$ writes to address $iaddr(c_{\mathrm{P}}^j)$, then an intermediate instruction $I(c_{\mathrm{P}}^k)$ for $i < k < j$ must drain the pipe. The same must hold if c_{P}^j is in user mode and $I(c_{\mathrm{P}}^i)$ writes to $ptea(c_{\mathrm{P}}^j, c_{\mathrm{P}}^j.DPC)$. Finally, mode can only be changed to user mode by an `rfe` (return from exception) instruction (and the hardware guarantees that `rfe` instructions drain the pipe).

Conditions of this nature are hypotheses of the hardware correctness proof in [12]. It will be easy to show that they hold for the kernels constructed in Sect. 6.

4 Compilation

We sketch the formal semantics of $C0$, a subset of C, and state the correctness theorem of a $C0$ compiler, summarizing result from [13]. In Section 4.3 we extend the $C0$ semantics to inline assembler code.

4.1 *C*0 Semantics

Eventually we want to consider several programs running under an operating system. The computations of these programs then are interleaved. Therefore our compiler correctness statement is based on a small steps / structured operational semantics [16,17].

In $C0$ types are elementary ($bool$, int, ...), pointer types, or composite ($array$ or $struct$). A type is called simple if it is an elementary type or a pointer type. We define the (abstract) size of types for simple types t by $size(t) = 1$, for arrays by $size(t[n]) = n \cdot size(t)$, and for structures by $size(struct\{n_1 : t_1, \dots, n_s : t_s\}) = \sum_i size(t_i)$. Values of variables with simple type are called *simple values*. Variables with composite types have *composite values* that are represented flat as a sequence of simple values.

Configuration. An $C0$ machine configuration c_{C0} has the following components:

1. The *program rest* $c_{\mathrm{C0}}.pr$. This is a sequence of $C0$ statements which still needs to be executed. In [16] the program rest is called *code component* of the configuration.
2. The *type table* $c_{\mathrm{C0}}.tt$ collects information about types used in the program.
3. The *function table* $c_{\mathrm{C0}}.ft$ contains information about the functions of a program. It maps function names f to pairs $c_{\mathrm{C0}}.ft(f) = (c_{\mathrm{C0}}.ft(f).ty, c_{\mathrm{C0}}.ft(f).body)$ where $c_{\mathrm{C0}}.ft(f).ty$ specifies the types of the arguments, the local variables, and the result of the function, whereas $c_{\mathrm{C0}}.ft(f).body$ specifies the function body.
4. The *recursion depth* $c_{\mathrm{C0}}.rd$.
5. The *local memory stack* $c_{\mathrm{C0}}.lms$. It maps numbers $i \leq c_{\mathrm{C0}}.rd$ to memory frames (defined below). The global memory is $c_{\mathrm{C0}}.lms(0)$. We denote the top local memory frame of a configuration c_{C0} by $top(c_{\mathrm{C0}}) = c_{\mathrm{C0}}.lms(c_{\mathrm{C0}}.rd)$.
6. A *heap memory* $c_{\mathrm{C0}}.hm$. This is also a memory frame.

Memory Frames. We use a relatively explicit, low level memory model in the style of [18]. Memory frames m have the following components: (i) the number $m.n$ of variables in m (for local memory frames this also includes the parameters of the corresponding function definition), (ii) a function $m.name$ mapping variable numbers $i \in [0 : m.n - 1]$ to their names (not used for variables on the heap), (iii) a function $m.ty$ mapping variable numbers to their type. This permits to define the size of a memory frame $size(m)$ as the number of simple values stored in it, namely: $size(m) = \sum_{i=0}^{m.n-1} size(m.ty(i))$. (iv) a content function $m.ct$ mapping indices $0 \leq i < size(m)$ to simple values.

A *variable* of configuration c_{C0} is a pair $v = (m, i)$ where m is a memory frame of c_{C0} and $i < m.n$ is the number of the variable in the frame. The type of a variable (m, i) is defined by $ty((m, i)) = m.ty(i)$.

Sub variables $S = (m, i)s$ are formed from variables (m, i) by appending a *selector* $s = (s_1, \ldots, s_t)$, where each component of a selector has the form $s_i = [j]$ for selecting array element number j or the form $s_i = .n$ for selecting the struct component with name n. If the selector s is consistent with the type of (m, i), then $S = (m, i)s$ is a *sub variable* of (m, i). Selectors are allowed to be empty. In $C0$, pointers p may point to sub variables $(m, i)s$ in the global memory or on the heap. The value of such pointers simply has the form $(m, i)s$. Component $m.ct$ stores the current values $va(c_{\mathrm{C0}}, (m, i)s)$ of the simple sub variables $(m, i)s$ in the canonical order. Values of composite variables x are represented in $m.ct$ in the obvious way by sequences of simple values starting from the abstract base address $ba(x)$ of variable x.

With the help of visibility rules and bindings we easily extend the definition of va, ty, and ba from variables and sub variables to expressions e.

Computation. For space restrictions we cannot give the definitions of the (small-step) transition function δ_{C0} mapping $C0$ configurations c_{C0} to their successor configuration $c'_{\mathrm{C0}} = \delta_{\mathrm{C0}}(c_{\mathrm{C0}})$. As an example we give a partial definition of the function call semantics.

Assume the program rest in configuration c_{C0} begins with a call of function f with parameters $e_1, \ldots, e_n$ assigning the function's result to variable v, formally $c_{\mathrm{C0}}.pr =$

$fcall(f, v, e_1, \ldots, e_n); r$. In the new program rest, the call statement is replaced by the body of function f taken from the function table, $c'_{\mathrm{C0}}.pr = c_{\mathrm{C0}}.ft(f).body; r$ and the recursion depth is incremented $c'_{\mathrm{C0}}.rd = c_{\mathrm{C0}}.rd+1$. Furthermore, the values of all parameters e_i are stored in the new top local memory frame by updating its content function at the corresponding positions: $top(c'_{\mathrm{C0}}).ct_{size(ty(c_{\mathrm{C0}},e_i))}(ba(c_{\mathrm{C0}}, e_i)) = va(c_{\mathrm{C0}}, e_i)$.

4.2 Compiler Correctness

The compiler correctness statement (with respect to physical machines) depends on a simulation relation $consis(aba)(c_{\mathrm{C0}}, c_{\mathrm{P}})$ between configurations c_{C0} of $C0$ machines and configurations c_{P} of physical machines which run the compiled program. The relation is parameterized by a function aba which maps sub variables S of the $C0$ machine to their allocated base addresses $aba(c_{\mathrm{C0}}, S)$ in the physical machine. The allocation function may change during a computation (i) if the recursion depth and thus the set of local variables change due to calls and returns or (ii) if reachable variables are moved on the heap during garbage collection (not yet implemented).

Simulation Relation. The simulation relation consists essentially of four conditions:

1. Value consistency $v\text{-}consis(aba)(c_{\mathrm{C0}}, c_{\mathrm{P}})$: this condition states, that reachable elementary sub variables x have the same value in the $C0$ machine and in the physical machine. Let $asize(x)$ be the number of bytes needed to store a value of type $ty(x)$. Then we require $c_{\mathrm{P}}.pm_{asize(x)}(aba(c_{\mathrm{C0}}, x)) = va(c_{\mathrm{C0}}, x)$.
2. Pointer consistency $p\text{-}consis(aba)(c_{\mathrm{C0}}, c_{\mathrm{P}})$: This predicate requires for reachable pointer variables p which point to a sub variable y that the value stored at the allocated address of variable p in the physical machine is the allocated base address of y, i.e. $c_{\mathrm{P}}.pm_4(aba(c_{\mathrm{C0}}, p)) = aba(c_{\mathrm{C0}}, y)$. This induces a sub graph isomorphism between the reachable portions of the heaps of the $C0$ and the physical machine.
3. Control consistency $c\text{-}consis(c_{\mathrm{C0}}, c_{\mathrm{P}})$: This condition states that the delayed PC of the physical machine (used to fetch instructions) points to the start of the translated code of the program rest $c_{\mathrm{C0}}.pr$ of the $C0$ machine. We denote by $caddr(s)$ the address of the first assembler instruction which is generated for statement s. We require $c_{\mathrm{P}}.DPC = caddr(c_{\mathrm{C0}}.pr)$ and $c_{\mathrm{P}}.PC = c_{\mathrm{P}}.DPC + 4$.
4. Code consistency $code\text{-}consis(c_{\mathrm{C0}}, c_{\mathrm{P}})$: This condition requires that the compiled code of the $C0$ program is stored in the physical machine c_{P} beginning at the code start address $cstart$. Thus it requires that the compiled code is not changed during the computation of the physical machine and thereby forbids self modifying code.

Theorem 2. *For every C0 machine computation $(c^0_{\mathrm{C0}}, c^1_{\mathrm{C0}}, \ldots)$ there are a computation $(c^0_{\mathrm{P}}, c^1_{\mathrm{P}}, \ldots)$ of the physical machine, step numbers $(s(0), s(1), \ldots)$, and a sequence of allocation functions $(aba^0, aba^1, \ldots)$ such that for all steps i and $S = s(i)$ we have $consis(aba^i)(c^i_{\mathrm{C0}}, c^S_{\mathrm{P}})$.*

4.3 Inline Assembler Code Semantics

For sequences u of assembler instructions (we do not distinguish here between assembler and machine language) we extend $C0$ by statements of the form $asm(u)$ and call

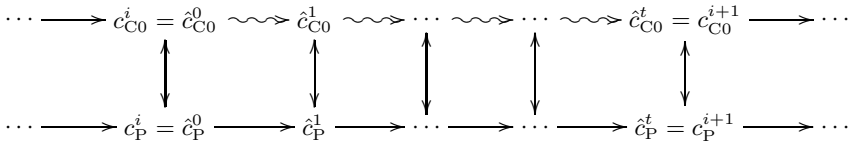


Fig. 2. Execution of Inline Assembler Code

the resulting language $C0_A$. In $C0_A$ the use of inline assembler code is restricted: (i) only a certain subset of DLX instructions is allowed (e.g. no load or store of bytes or half words, only *relative* jumps), (ii) the target address of store word instructions must be outside the code and data regions of the $C0_A$ program or it must be equal to the allocated base address of a sub variable of the $C0_A$ program with type int or $unsigned\ int$ (this implies that inline assembler code cannot change the stack layout of the $C0_A$ program), (iii) the last assembler instruction in u must not be a jump or branch instruction, (iv) the execution of u must terminate, (v) the target of jump and branch instructions must not be outside the code of u, and (vi) the execution of u must not generate misalignment or illegal instruction interrupts.

As pointed out in Sect. 2, operating system kernels necessarily contain assembler code; thus a formal semantics of programs in $C0_A$ has to be defined. Inline assembler portions of $C0_A$ programs can modify parts of the machine which are not visible to $C0$, e.g. the processor registers or memory which is not reachable via $C0$ variables. Thus to define the meaning of inline assembler code the transition function δ_{C0A} of $C0_A$ machines needs, in addition to a $C0$ configuration c_{C0}, a physical machine configuration as a second input parameter. Also the result of δ_{C0A} consists of a $C0$ configuration *and* a physical machine configuration. To express the meaning of inline assembler code which changes memory cells holding $C0$ variables we parameterize the $C0_A$ transition function over an allocated base address function aba like we did for the $consis$ relation.

As long as no inline assembler code is executed, we set $\delta_{\mathrm{C0A}}(aba)(c^i_{\mathrm{C0}}, c^i_{\mathrm{P}}) = (\delta_{\mathrm{C0}}(c^i_{\mathrm{C0}}), x)$ ignoring the second input parameter and setting the second output parameter to an arbitrary, fixed physical machine configuration x.

However, when executing inline assembler code, $c^i_{\mathrm{C0}}.pr = asm(u); r$, the definition of $\delta_{\mathrm{C0A}}(aba)(c^i_{\mathrm{C0}}, c^i_{\mathrm{P}}) = (c^{i+1}_{\mathrm{C0}}, c^{i+1}_{\mathrm{P}})$ is more difficult. We take c_{P} as the start configuration for the execution of assembler code sequence u. The execution of u leads to a physical machine computation $(\hat{c}^0_{\mathrm{P}}, \ldots, \hat{c}^t_{\mathrm{P}})$ with $\hat{c}^t_{\mathrm{P}}.DPC = caddr(r)$ and $\hat{c}^t_{\mathrm{P}}.PC = \hat{c}^t_{\mathrm{P}}.DPC + 4$ by the restrictions on inline assembler. We construct a corresponding sequence $(\hat{c}^0_{\mathrm{C0}}, \ldots, \hat{c}^t_{\mathrm{C0}})$ of intermediate $C0$ machine configurations reflecting successively the possible updates of the C0 variables by the assembler instructions (see Fig. 2). We set $\hat{c}^0_{\mathrm{C0}} = c^i_{\mathrm{C0}}$ except for the program rest: $\hat{c}^0_{\mathrm{C0}}.pr = r$. If the instruction $I(\hat{c}^j_{\mathrm{P}})$ executed in configuration $\hat{c}^j_{\mathrm{P}}$ for $j < t$ writes the value v to the word at an address $ea(\hat{c}^j_{\mathrm{P}})$ equaling the allocated base address of some $C0$ variable x, we update the corresponding variable in the $\hat{c}^{j+1}_{\mathrm{C0}}$ by $va(\hat{c}^{j+1}_{\mathrm{C0}}, x) = v$. Finally the result of the $C0_A$ transition function is defined by $c^{i+1}_{\mathrm{C0}} = \hat{c}^t_{\mathrm{C0}}$ and $c^{i+1}_{\mathrm{P}} = \hat{c}^t_{\mathrm{P}}$.

Observe that for the definition from above we do not need to know c^i_{P} exactly. Nevertheless the definition keeps configurations consistent:

Lemma 1. *If the program rest of c^i_{C0} starts with an inline assembler statement we have* $consis(aba)(c^i_{\mathrm{C0}}, c^i_{\mathrm{P}}) \implies consis(aba)(\delta_{\mathrm{C0A}}(aba)(c^i_{\mathrm{C0}}, c^i_{\mathrm{P}}))$.

5 CVM Semantics

We introduce communicating virtual machines (CVM), a model of computation for a generic abstract operating system kernel interacting with a fixed number of user processes. CVM uses the $C0$ language semantics to model computation of the (abstract) kernel and virtual machines to model computation of the user processes. It is a pseudo-parallel model in the sense that in every step of computation either the kernel or one user process can make progress.

From a kernel implementor's point of view, CVM encapsulates the low-level functionality of a microkernel and provides access to it as a library of functions, the so-called CVM primitives. Accordingly, the abstract kernel may be 'linked' against the implementation of these primitives to produce the concrete kernel, a $C0_A$ program, that may be run on the target machine. This construction and its correctness will be treated in Sect. 6. In the following sections we define CVM configurations, CVM computations, and show how abstract kernels implement system calls as regular $C0$ function calls.

5.1 Configurations

A CVM configuration c_{CVM} has the following components:

- User process virtual machine configurations $c_{\mathrm{CVM}}.up(u)$ for user process indices $u \in \{1, \ldots, P\}$ (and P fixed, e.g. $P = 128$).
- A $C0$ machine configuration $c_{\mathrm{CVM}}.ca$ of the so-called *abstract kernel*. As we will see below, the kernel configuration, in particular its initial configuration, must have a certain form: (i) it must have a global variable named i of type int, (ii) it declares certain functions $f \in CVMP$, the CVM primitives, with empty body, arguments, and effects as described below, and (iii) it must have a function $kdispatch$ that takes two integer arguments and returns an integer; when starting with a call to $kdispatch$ as initial program rest the kernel must eventually call $start$, a CVM primitive that passes control to one of the user processes.
- The component $c_{\mathrm{CVM}}.cp$ denotes the current process: $c_{\mathrm{CVM}}.cp = 0$ means that the kernel is running; $c_{\mathrm{CVM}}.cp = u > 0$ means that user process u is running.

5.2 Computation

A computation of the CVM machine is parameterized over a list of external interrupt events $eevs$, one event mask eev^e with e signals for each user process step (the kernel runs uninterruptibly).

In this section we define the next state function δ_{CVM} of the CVM model. It maps the external events' list $eevs$ and a CVM configuration c_{CVM} to its successor configurations c'_{CVM} and the new external events' list $eevs'$, so $\delta_{\mathrm{CVM}}(eevs, c_{\mathrm{CVM}}) = (c'_{\mathrm{CVM}}, eevs')$. In the definitions below we only list components that are changed.

User Computation. If the current process $c_{\mathrm{CVM}}.cp$ in configuration c_{CVM} is non-zero then user process $u = c_{\mathrm{CVM}}.cp$ is meant to make a step. Let $eevs = eev; eevs'$, i.e. $eev \in \{0,1\}^e$ denotes the first element of the external events' list and $eevs'$ its remainder, the next external events' list.

Let the predicate $JISR(c_{\mathrm{V}}, eev)$ denote that an interrupt occurred in configuration c_{V}, either internally or with respect to the events eev. If $JISR(c_{\mathrm{V}}, eev)$, then the (masked) exception cause is encoded in the bitvector $mca(c_{\mathrm{V}}, eev)$ and an additional 'parameter' of the exception (for internal exceptions only) is denoted by $edata(c_{\mathrm{V}})$. For details on the definition of $JISR$, mca, and $edata$ see e.g. [12,15].

For $\neg JISR(c_{\mathrm{CVM}}.up(u), eev)$ a CVM step simply consists of a step of the virtual machine $c_{\mathrm{CVM}}.up(u)$, so $c'_{\mathrm{CVM}}.up(u) = \delta_{\mathrm{V}}(c_{\mathrm{CVM}}.up(u))$. Otherwise, execution of the abstract kernel starts. The kernel's entry point is the function $kdispatch$ that is called with the exception masked cause and the exception data. We set the current process component and the kernel's recursion depth to zero, $c'_{\mathrm{CVM}}.cp = 0$ and $c'_{\mathrm{CVM}}.ca.rd = 0$, and the kernel's program rest to the function call $c'_{\mathrm{CVM}}.ca.pr = fcall(kdispatch, i, mca(c_{\mathrm{V}}, eev), edata(c_{\mathrm{V}}))$.

Kernel Computation. Initially (after power-up) and after an interrupt, as seen above, the kernel starts execution with a call of the function $kdispatch$. User process execution continues when the kernel calls the $start$ CVM primitive.

If we have $c_{\mathrm{CVM}}.cp = 0$ and the kernel's program rest does not start with a call to a CVM primitive, a regular $C0$ semantics step is performed, $c'_{\mathrm{CVM}}.ca = \delta_{\mathrm{C0}}(c_{\mathrm{CVM}}.ca)$.

Otherwise, we have $c_{\mathrm{CVM}}.cp = 0$ and $c_{\mathrm{CVM}}.cp.pr = fcall(f, v, e_1, \ldots, e_n); r$ for a CVM primitive f, an integer variable v and integer expressions e_1 to e_n. The CVM primitive $f = start$ to start user processes is defined below. For $f \neq start$, the CVM primitive f is specified by a function f_{S} that takes n integer arguments, a P-tuple of virtual machines and returns an integer and an updated P-tuple of virtual machines. The new CVM configuration after calling such a primitive is defined as follows. First, we compute the application of f_{S} to the values $E_i = va(c_{\mathrm{CVM}}.ca, e_i)$ of the expressions e_i and set $(v_{\mathrm{S}}, c'_{\mathrm{CVM}}.up) = f_{\mathrm{S}}(E_1, \ldots, E_n, c_{\mathrm{CVM}}.up)$. Then, the program rest of the kernel is set to $c'_{\mathrm{CVM}}.pr = r$ and the return value v_{S} is stored in the return variable v of the call to the CVM primitive.

Below we describe the special CVM primitive $start$ and then a few selected other primitives. For lack of space, we ignore any pre conditions or corner cases; these are straightforward to specify and resolve.

- The CVM primitive $start$, taking one argument, hands control over to the specified user process. For $c_{\mathrm{CVM}}.ca.pr = fcall(start, v, e_1); r$ and $u = va(c_{\mathrm{CVM}}.ca, e_1)$ we set $c'_{\mathrm{CVM}}.cp = u$. By this definition, the kernel stops execution and is restarted again on the next interrupt (with a fresh program rest as described before).
- The CVM primitive $alloc$ increases the memory size of user process u by x pages. We define $alloc_{\mathrm{S}}(u, x, c_{\mathrm{CVM}}.up) = (0, c'_{\mathrm{CVM}}.up)$ by increasing the memory size, $c'_{\mathrm{CVM}}.up(u).V = c_{\mathrm{CVM}}.up(u).V + x$, and afterwards clearing the new pages, $c'_{\mathrm{CVM}}.up(u).vm(y) = 0^8$ for $c_{\mathrm{CVM}}.up(u).V \cdot 4\mathrm{K} \leq y < c'_{\mathrm{CVM}}.up(u).V \cdot 4\mathrm{K}$.
- Likewise, the CVM primitive $free$ with specification function $free_{\mathrm{S}}$ decreases the memory size of a user process u by x pages.

- The CVM primitive *copy* copies memory between user processes. So we define $copy_{\mathrm{S}}(u_1, a_1, u_2, a_2, d, c_{\mathrm{CVM}}.up) = (0, c'_{\mathrm{CVM}}.up)$ by $c'_{\mathrm{CVM}}.up(u_2).vm_d(a_2) = c_{\mathrm{CVM}}.up(u_1).vm_d(a_1)$.
- The CVM primitive *get_vm_gpr* reads register $GPR[r]$ of process u; we define $get\_vm\_gpr_{\mathrm{S}}(r, u, c_{\mathrm{CVM}}.up) = (c_{\mathrm{CVM}}.up(u).GPR[r], c_{\mathrm{CVM}}.vm)$. As described below, this primitive is used to read parameters of system calls.
- The CVM primitive *set_vm_gpr* writes register $GPR[r]$ of process u; we define $set\_vm\_gpr_{\mathrm{S}}(r, u, x, c_{\mathrm{CVM}}.up) = (0, c'_{\mathrm{CVM}}.up)$ by $c'_{\mathrm{CVM}}.up(u).GPR[r] = x$. This primitive is used to set return values of system calls.

The remaining CVM primitives include process initialization (*reset* and *clone*) or device port I/O (*input* and *output*).

5.3 Abstract Kernels and System Calls

The binary interface of a kernel specifies how user processes can make system calls to the kernel. We describe an exemplary binary interface, also used in the VAMOS kernel [10]: a system call number j is invoked by a trap instruction with immediate constant j. System calls have additional parameters that are taken from general purpose registers of the user process; if system call j has n parameters we pass parameter number x with $1 \leq x \leq n$ in register $GPR[10 + x]$ of the calling process. Furthermore, after completion of the system call the kernel notifies the user process of the result of the system call by updating a return value register, e.g. $GPR[20]$, of the calling process.

In a CVM based kernel such a system call interface is implemented as follows. Let the kernel maintain a variable cu that indicates the last process that has been started. Execution of a trap instruction with immediate constant j causes an interrupt with index 5. In the absence of other higher-prioritized interrupts, this interrupt entails a function call $kdispatch(mca, j)$ in the abstract kernel with $mca[5:0] = 100000$ that the kernel then detects as a system call j of process cu. Testing the parameter j the kernel determines the number of parameters n and a function f that is meant to handle the system call. It calls the *get_vm_gpr* CVM primitive repeatedly for all $1 \leq x \leq n$ with $fcall(get\_vm\_gpr, e_x, cu, x)$ to set the parameters of the call such that $e_x = c_{\mathrm{CVM}}.up(cu).GPR[10 + x]$. Then, the actual call of the handler is implemented as an ordinary $C0$ function call $fcall(f, r, e_1, \ldots, e_n)$ in the abstract kernel. The return result is passed back to the user ($fcall(set\_vm\_gpr, i, r)$) and the user process is reactivated ($fcall(start, i, cu)$). We see that not only the semantics but also the implementation of a trap interrupt is formally a function call.

6 Concrete Kernels and Correctness

The concrete kernel cc is an implementation of the CVM model for a given abstract kernel ca. We construct the concrete kernel by *linking* the abstract kernel ca, a $C0$ program, with a CVM implementation cvm, a $C0_A$ program. Formally, this is written using a link operator ld as $cc = ld(ca, cvm)$. The function table of the linked program cc is constructed from the function tables of the input programs. For functions present in both programs, *defined functions* (with a non-empty body) take precedence over

declared functions (without a body). We do not formally define the ld operator here; it may only be applied under various restrictions concerning the input programs, e.g. the names of global variables of both programs must be distinct, function signatures must match, and no function may be defined in both input programs. We require that the abstract kernel ca defines $kdispatch$ and declares all CVM primitives while the CVM implementation cvm defines the primitives and declares $kdispatch$.

6.1 CVM Implementation

Data Structures. The CVM implementation maintains data structures for the simulation of the virtual machines and multiprocessing. These include: (i) An array of process control blocks $pcb[u]$ for the kernel ($u = 0$) and the user processes ($u > 0$). Process control blocks are structures with components $pcb[u].R$ for every processor register R of the *physical* machine. (ii) The integer array $ptspace$ on the heap holds the page tables of all user processes. Its base address must be a multiple of 4K. (iii) Data structures (e.g. doubly-linked lists) for the management of physical and swap memory (including victim selection for page faults). (iv) The variable cup keeping track of the current user process thus encoding the $c_{\mathrm{CVM}}.cp$ component.

Entering System Mode. If the concrete kernel enters system mode, its program rest is initialized with $init_1; init_2$. In all other cases than reset, the first part $init_1$ will (i) write all processor register R to the process control block $PCB[cup].R$ of the process cup that was interrupted and (ii) restore the registers of the kernel from process control block $PCB[0]$. Only after the execution of $init_1$, compiler consistency holds. In the second part $init_2$, the CVM implementation detects whether the interrupt was due to a page fault or for other causes. Page faults are handled silently without calling the abstract kernel (cf. below). For other interrupts, we call $kdispatch$ with parameters obtained from $PCB[cup]$.

Leaving System Mode. The $start$ CVM primitive enters user mode again. It is implemented using inline assembler. First, we assign the parameter u of $start$ to cup. Second, we write the physical processor registers to $PCB[0]$ to save the concrete kernel state. Third, we restore the physical processor registers for process u from $PCB[u]$ and execute an `rfe` (return from exception).

Page Fault Handler. The page fault handler establishes the simulation relation B as described in [12] and summarized in Sect. 3. Only with correct page fault handlers, user mode steps in the physical machine without interrupts simulate steps of a virtual machine. Again, note that a user mode instruction can produce up to two page faults.

To reason about multiple user processes u, we have to slightly modify and extend the B relation. Have a virtual machine configuration c_{V} and a physical machine configuration c_{P}. If user process u is not running, i.e. $cup \neq u$ or $c_{\mathrm{P}}.mode = 0$, we demand that the user-visible processor registers of the process *and* the location and size of its page table (via special-purpose registers pto and ptl) are stored in the process control block $pcb[u]$. Thus, we parameterize B over user processes u and set $B(u)(c_{\mathrm{V}}, c_{\mathrm{P}}) = B(c_{\mathrm{V}}, \hat{c}_{\mathrm{P}})$ where $\hat{c}_{\mathrm{P}}$ is defined by $\hat{c}_{\mathrm{P}}.m = c_{\mathrm{P}}.m$ for $m \in \{pm, sm\}$ and $\hat{c}_{\mathrm{P}}.R = c_{\mathrm{P}}.R$ if $cup = u$ and $c_{\mathrm{P}}.mode = 1$ or $\hat{c}_{\mathrm{P}}.R = pcb[u].R$ otherwise.

Implementation of the CVM Primitives. The implementation of CVM primitives like get_vm_gpr and set_vm_gpr is straightforward with the entry and exit mechanism updating the process control blocks described before. For CVM primitives $alloc$ and $free$ the page table length of the process has to to be increased or decreased, resp.; various other data structures concerning memory management have to be adjusted as well. Such operations are closely interconnected with the page fault handler. Since the page tables are accessible as a C0 data structure, inline assembler is only required to clear physical pages. Similarly, the $copy$ implementation requires assembler to copy physical pages.

6.2 Simulation Relation Between Abstract Kernel and Concrete Kernel

We proceed as in Sect. 4.2 by defining a simulation relation $konsis(kalloc)(cc, ca)$ that states whether a concrete kernel configuration cc encodes an abstract kernel configuration ca; this relation is parameterized over a function $kalloc$ mapping variables in the abstract to variables in the concrete kernel. Note that the concrete kernel may has more variables than the abstract kernel (i) in the global memory frame $cc.lms(0)$ and (ii) on the heap $cc.hm$.

By placing the additional global variables *behind* the global variables of the abstract kernel, indices of variables from the abstract kernel stay unchanged for any memory frame $lms(i)$. Hence, we define $kalloc(ca.lms(i), j) = (cc.lms(i), j)$. Heap variables $(ca.hm, j)$ in the abstract kernel must be mapped injectively to heap variables $kalloc(ca.hm, j) = (cc.hm, j')$ in the concrete kernel. Below we demand that the abstract heap is embedded isomorphically in the concrete heap. For sub variables Vs, we trivially extend $kalloc(Vs) = kalloc(V)s$. Now we set $konsis(kalloc)(cc, ca)$ iff (i) program rests and recursion depths coincide, $cc.pr = ca.pr$ and $cc.rd = ca.rd$, (ii) corresponding elementary sub variables S and $kalloc(S)$ have the same value, $va(ca, S) = va(cc, kalloc(S))$, (iii) reachable pointer sub variables P and $kalloc(P)$ must point to corresponding locations, $kalloc(va(ca, P)) = va(cc, kalloc(P))$. For pointers P to heap variables, i.e. $va(ca, P) = (ca.hm, j)$, this establishes a sub graph isomorphism between the heaps of the abstract and the concrete kernel.

6.3 Correctness of the Concrete Kernel

Our formulation of a correctness theorem for an operating system kernel written in $C0$ uses the result for virtual memory simulation (Section 3), the compiler correctness theorem (Section 4), Lemma 1 on the execution of inline assembler (Section 4.3), and the simulation relation between abstract kernels and concrete kernels (Section 6).

Consider an initial CVM configuration c^0_{CVM} with a valid abstract kernel configuration $c^0_{\mathrm{CVM}}.ca$. Let $cc^0 = ld(c^0_{\mathrm{CVM}}.ca, cvm)$ denote the initial configuration of the concrete kernel and let $(c^0_{\mathrm{P}}, c^1_{\mathrm{P}}, \dots)$ denote a physical machine computation with c^0_{P} code-consistent to cc^0. After $z(0)$ steps the physical machine is fully consistent to cc^0 under an allocated base address function aba^0, i.e. $consis(aba^0)(cc^0, c^{z(0)}_{\mathrm{P}})$.

We construct the list of external events $eevs_{\mathrm{CVM}} = (eev^0_{\mathrm{CVM}}, eev^1_{\mathrm{CVM}}, \dots)$ that parameterizes the CVM computation based on the external event signals eev^k_{P} seen by the physical processor in step k. We sample the external events for non-page-faulting user mode steps. Formally, let the sequence $x(l)$ enumerate these steps ascendingly and

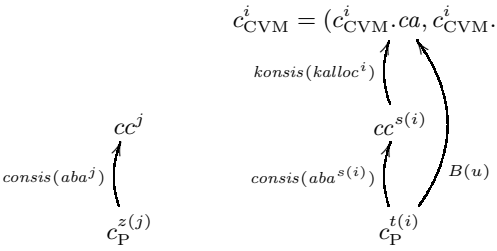


Fig. 3. Consistency Relations Between the Different Configurations

define $eev^l_{\mathrm{CVM}} = eev^{x(l)}_{\mathrm{P}}$. Hence, page faults may 'shadow' events with respect to the CVM; this problem must be treated when fully specifying I/O devices.

Let $(c^0_{\mathrm{CVM}}, c^1_{\mathrm{CVM}}, \dots)$ denote the CVM computation parameterized over the external events' list $eevs_{\mathrm{CVM}}$. Then, there exists (i) a sequence of concrete kernel configurations $(cc^0, cc^1, \dots)$, (ii) a sequence of step numbers $z(j)$ and allocated base address functions aba^j relating the physical machine computation to the sequence of concrete kernel configurations, and (iii) a sequence of step numbers $s(i)$ and functions $kalloc^i$ relating the abstract kernel's computation to the sequence of concrete kernel configurations such that for all i and j the following simulation relations hold:

- Configuration $cc^{s(i)}$ of the concrete kernel after step $s(i)$ encodes configuration $c^i_{\mathrm{CVM}}.ca$ of the abstract kernel after step i, so $konsis(kalloc^i)(c^i_{\mathrm{CVM}}.ca, cc^{s(i)})$.
- Configuration $c^{z(j)}_{\mathrm{P}}$ of the physical machine after step $z(j)$ encodes configuration cc^j of the concrete kernel after step j, so $consis(aba^j)(cc^j, c^{z(j)}_{\mathrm{P}})$.
- For all user processes u the configuration $c^i_{\mathrm{CVM}}.up(u)$ of virtual machine u after step i of the CVM machine is encoded by the configuration $c^{t(i)}_{\mathrm{P}}$ of the physical machine after step $t(i) = z(s(i))$. With the B relation introduced in Sect. 3 and parameterized in Sect. 6, we require $B(u)(c^i_{\mathrm{CVM}}.up(u), c^{t(i)}_{\mathrm{P}})$.
- The physical machine computation and the computation of the concrete kernel must fit together. Unless $cc^j.pr = asm(u, \texttt{rfe}); r$, i.e. the program rest starts with assembler code that returns to user mode, cc^{j+1} is computed by δ_{C0A} parameterized with the current allocated base address function aba^j applied to the current physical machine configuration $c^{z(j)}_{\mathrm{P}}$ and cc^j. Formally, $(cc^{j+1}, c^{z(j+1)}_{\mathrm{P}}) = \delta_{\mathrm{C0A}}(aba^j)(cc^j, c^{z(j)}_{\mathrm{P}})$. Observe that the aba parameter and the second input for δ_{C0A} are used only for executing inline assembler code. In the other case, i.e. if $cc^j.pr = asm(u, \texttt{rfe}); r$, the next configuration of the concrete kernel cc^{j+1} is obtained from cc^j by setting $cc^{j+1}.pr = init_2$ and $cc^{j+1}.rd = 1$.

The claim of the correctness theorem is illustrated in Fig. 3. Its proof is by induction on i with a case split along the cases of the CVM semantics from Sect. 5. In the proof the sequence numbers $z(j)$ and $s(i)$ are defined inductively: (i) Unless $cc^j.pr = asm(u, \texttt{rfe}); r$ we set $z(j+1)$ as in the induction step of the compiler correctness theorem. If the program rest starts with $asm(u, \texttt{rfe})$ just before returning to user mode, we set $z(j+1)$ to the index of that that system mode configuration that marks the completed initialization part $init_1$ of the concrete kernel. This resembles the base case

of compiler correctness. (ii) We set $s(i+1) = s(i)$ if $c^i_{\mathrm{CVM}}.cp \neq 0$ or $c^i_{\mathrm{CVM}}.pr = fcall(start, v, e_1); r$, $s(i+1) = s(i)+1$ if $c^i_{\mathrm{CVM}}.cp = 0$ and if $c^i_{\mathrm{CVM}}.ca.pr$ does not start with a CVM primitive call. In this case, the concrete kernel simulates one step of the abstract. We set $s(i+1) = s(i)+x$ if $c^i_{\mathrm{CVM}}.cp = 0$ and if the program rest $c^i_{\mathrm{CVM}}.ca.pr$ starts with a call of a CVM primitive other than $start$, and x is the number of steps the implementation of the CVM primitive takes to return.

7 Status of the Formal Verification

At the time of this writing a considerable part of the presented work has been formalized in the theorem prover Isabelle/HOL [19]: (i) based on the specification of the VAMP processor [12,20] (a DLX-like processor verified in PVS) we have specified a formal VAMP assembler semantics, (ii) we have defined formal semantics for $C0$ and $C0_A$, (iii) we specified as an Isabelle/HOL function, implemented in $C0$, and verified the compiler's code generation, (iv) large parts of the compiler simulation theorem (Sect. 4.2) have been verified (we plan to finish this proof until fall 2005), (v) data structures and algorithm used in the CVM implementation have been specified and verified, (vi) the CVM and the VAMOS microkernel semantics have been formally specified.

8 Summary and Further Work

The work presented here depends crucially on a recent theory of virtual memory simulation from [12] and a compiler correctness proof in form of a step by step simulation theorem from [13]. We have presented the new abstract CVM model. In this model the formalisms for machine language specification and for programming language semantics have been combined in a natural way, allowing to treat system calls not only intuitively but also formally as function calls. Also, due to the parallelism, CVM permits to specify operating system kernel without reference to inline assembler code. We have provided an approach to the pervasive verification of an operating system kernel, which handles virtual memory and is written in a high level language with inline assembler code, and outlined substantial parts of its proof.

The 'trivial' further work is the completion of the formal verification effort that we expect to be completed in 2006 if things go well or in 2007 if things go not so well. In the next years CVM will be used in the Verisoft project [10] in several places: (i) a correctness proof for a simple operating system (called SOS) will be based on CVM with a particular abstract kernel (called VAMOS) inspired by [6]. (ii) Based on verified hardware [12,20], verified compilers, and SOS the verification of entire systems for electronic signatures of emails and for biometric access control will be attempted.

References

1. Hutt, A.E., Hoyt, D.B., Bosworth, S., eds.: Computer Security Handbook. John Wiley & Sons, Inc., New York, NY, USA (1995)
2. Shapiro, J.S., Hardy, N.: Eros: A principle-driven operating system from the ground up. IEEE Software **19** (2002) 26–33

3. Rushby, J.: Proof of separability: A verification technique for a class of security kernels. In: Proc. 5th International Symposium on Programming, Turin, Italy, Springer (1982) 352–367
4. Wulf, W.A., Cohen, E.S., Corwin, W.M., Jones, A.K., Levin, R., Pierson, C., Pollack, F.J.: HYDRA: The kernel of a multiprocessor operating system. CACM **17** (1974)
5. Pfitzmann, B., Riordan, J., Stüble, C., Waidner, M., Weber, A.: The PERSEUS system architecture. In Fox, D., Köhntopp, M., Pfitzmann, A., eds.: VIS 2001, Sicherheit in komplexen IT-Infrastrukturen, Vieweg Verlag (2001) 1–18
6. Liedtke, J.: On micro-kernel construction. In: Proceedings of the 15th ACM Symposium on Operating systems principles, ACM Press (1995) 237–250
7. The Common Criteria Project Sponsoring Organisations: Common Criteria for Information Technology Security Evaluation version 2.1, Part I. `http://www.commoncriteriaportal.org/public/files/ccpart1v21.pdf` (1999)
8. Bevier, W.R.: Kit: A study in operating system verification. IEEE Transactions on Software Engineering **15** (1989) 1382–1396
9. OSEK group: OSEK/VDX time-triggered operating system. `http://www.osek-vdx.org/mirror/ttos10.pdf` (2001)
10. The Verisoft Consortium: The Verisoft project. `http://www.verisoft.de/` (2003)
11. Aho, A.V., Hopcroft, J.E., Ullman, J.: Data Structures and Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
12. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. Technical report, Verisoft project (2005) `http://www.verisoft.de/.rsrc/SubProject2/verificationmm.pdf`.
13. Leinenbach, D., Paul, W., Petrova, E.: Compiler verification in the context of pervasive system verification. Draft manuscript (2005)
14. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Second edn. Morgan Kaufmann, San Mateo, CA (1996)
15. Müller, S.M., Paul, W.J.: Computer Architecture: Complexity and Correctness. Springer (2000)
16. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. John Wiley & Sons, Inc., New York, NY, USA (1992, revised online version: 1999)
17. Winskel, G.: The formal semantics of programming languages. The MIT Press (1993)
18. Norrish, M.: C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory (1998)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of Lecture Notes in Computer Science (LNCS). Springer (2002)
20. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP processor. In Geist, D., Tronci, E., eds.: CHARME '03, Springer (2003) 51–65

Alpha-Structural Recursion and Induction
(Extended Abstract)

Andrew M. Pitts

University of Cambridge Computer Laboratory, Cambridge CB3 0FD, UK

Abstract. There is growing evidence for the usefulness of name *permutations* when dealing with syntax involving names and name-binding. In particular they facilitate an attractively simple formalisation of common, but often technically incorrect uses of structural recursion and induction for abstract syntax trees modulo α-equivalence. At the heart of this formalisation is the notion of *finitely supported* mathematical objects. This paper explains the idea in as concrete a way as possible and gives a new derivation within higher-order logic of principles of α-*structural* recursion and induction for α-equivalence classes from the ordinary versions of these principles for abstract syntax trees.

1 Introduction

> "They [previous approaches to operational semantics] do not in general have any great claim to being *syntax-directed* in the sense of defining the semantics of compound phrases in terms of the semantics of their components."
>
> —GD Plotkin, *A Structural Approach to Operational Semantics*, p 21 (Aarhus, 1981; reprinted as [18, p 32])

The above quotation and the title of the work from which it comes indicate the important role played by *structural recursion* and *structural induction* in programming language semantics. These are the forms of recursion and induction that fit the commonly used "algebraic" treatment of syntax. In this approach one specifies the syntax of a language at the level of abstract syntax trees (ASTs) by giving an *algebraic signature*. This consists of a collection of *sorts* s (one for each syntactic category of the language), and a collection of *constructors* K (also called "operators" or "function symbols"). Each such K comes with an *arity* consisting of a finite list $(\mathsf{s}_1, \ldots, \mathsf{s}_n)$ of sorts and with a *result-sort* s. Then the ASTs over the signature can be described by inductively generated terms t: if K has arity $(\mathsf{s}_1, \ldots, \mathsf{s}_n)$ and result sort s, and if t_i is a term of sort s_i for $i = 1..n$, then $K(t_1, \ldots, t_n)$ is a term of sort s. One gets off the ground in this inductive definition with the $n = 0$ instance of the rule for forming terms; this covers the case of *constants*, C, and one usually writes the term $C()$ just as C. Recursive definitions and inductive proofs about programs following the structure of their ASTs are both clearer and less prone to error than ones using non-structural methods. However, this treatment of syntax does not take into account the fact that most languages that one deals with in programming semantics involve *binding* constructors. In the presence of binders many syntax-manipulating operations only make sense, or at least only have

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 17–34, 2005.

good properties, when we operate on syntax at a level of abstraction represented not by ASTs themselves, but by α-equivalence classes of ASTs.

It is true that this level of abstraction, which identifies terms differing only in the names of bound entities, can be reconciled with an algebraic treatment of syntax by using de Bruijn indexes [4]. The well-known disadvantage of this device is that it necessitates a calculus of operations on de Bruijn indexes that does not have much to do with our intuitive view of the structure of syntax. As a result there can be a big "coding gap" between statements of results involving binding syntax we would like to make and their de Bruijn versions; and (hence) it is easy to get the latter wrong. For this reason, de Bruijn-style representations of syntax may be more suitable for language implementations than for work on language semantics.

In any case, most of the work on semantics which is produced by humans rather than by computers sticks with ordinary ASTs involving explicit bound names and uses an informal approach to α-equivalence classes.[1] This approach is signalled by a form of words such as "we identify expressions up to α-equivalence" and means that: (a) occurrences of "t" now really mean its α-equivalence class "$[t]_\alpha$"; and (b) if the representative t for the class $[t]_\alpha$ is later used in some context where the particular bound names of t clash in some way with those in the context, then t will be changed to an α-variant whose bound names are fresh (i.e. ones not used in the current context). In other words it is assumed that the "Barendregt variable convention" [1, Appendix C] is maintained dynamically. In the literature, the ability to change bound names "on the fly" is usually justified by the assertion that final results of constructions involving ASTs are independent of choice of bound names. A fully formal treatment has to prove such independence results and in this paper we examine ways, arising from the results of [8, 16], to reduce the burden of such proofs.

However, proving that pre-existing functions respect α-equivalence is only part of the story; in most cases a prior (or simultaneous) problem is to prove the existence of the required functions in the first place. To see why, consider the familiar example of *capture-avoiding substitution* $(x := t)t'$ of a λ-term t for all free occurrences of a variable x in a λ-term t'. In the vernacular of programming semantics, we specify $(x := t)(-)$ by saying that it has the properties

$$(x := t)x_1 = \begin{cases} t & \text{if } x_1 = x \\ x_1 & \text{if } x_1 \neq x \end{cases} \tag{1}$$

$$(x := t)(t_1\, t_2) = (x := t)t_1\, (x := t)t_2 \tag{2}$$

$$(x := t)\lambda x_1.\, t_1 = \lambda x_1.\, (x := t)t_1 \quad \text{if } x_1 \neq x \text{ and } x_1 \text{ is not free in } t \tag{3}$$

where in the last clause there is no need to say what happens when $x_1 = x$ or when x_1 does occur freely in t, since we are working "up to α-equivalence" and can change $\lambda x_1.\, t_1$ to an α-variant satisfying these conditions. To see what this specification really amounts to, let us restore the usually-invisible notation for α-equivalence classes. Writing Λ for the set of λ-terms and $\Lambda/{=_\alpha}$ for its quotient by α-equivalence $=_\alpha$, then capture-avoiding substitution of $e \triangleq [t]_\alpha$ for x is a function $\hat{s}_{x,e} \in \Lambda/{=_\alpha} \to \Lambda/{=_\alpha}$.

[1] This includes the metatheory of "higher-order abstract syntax" [15], where the questions we are addressing are pushed up one meta-level to a single binding-form, λ-abstraction.

Every such function corresponds to a function $s_{x,e} \in \Lambda \to \Lambda/{=_\alpha}$ respecting $=_\alpha$, i.e. satisfying

$$t_1 =_\alpha t_2 \;\Rightarrow\; s_{x,e}(t_1) = s_{x,e}(t_2) \tag{4}$$

(enabling us to define $\hat{s}_{x,e}([t']_\alpha)$ as $[s_{x,e}(t')]_\alpha$). The requirements (1)–(3) mean that we want $s_{x,e}$ to satisfy:

$$s_{x,e}(x_1) = \begin{cases} e & \text{if } x_1 = x \\ [x_1]_\alpha & \text{if } x_1 \neq x \end{cases} \tag{5}$$

$$s_{x,e}(t_1\, t_2) = [t_1'\, t_2']_\alpha \quad \text{where } s_{x,e}(t_i) = [t_i']_\alpha \text{ for } i = 1, 2 \tag{6}$$

$$s_{x,e}(\lambda x_1.t_1) = [\lambda x_1.t_1']_\alpha \quad \begin{array}{l}\text{if } x_1 \neq x \text{ and } x_1 \text{ is not free in } e, \\ \text{and where } s_{x,e}(t_1) = [t_1']_\alpha.\end{array} \tag{7}$$

The problem is not one of proving that a certain well-defined function $s_{x,e}$ respects α-equivalence, but rather of proving that a function exists satisfying (4)—(7). Note that (5)—(7) do not constitute a definition of $s_{x,e}(t')$ by recursion on the structure of the AST t': even if we patch up the conditions in clauses (6) and (7) by using some enumeration of ASTs to make the choices t_i' definite functions of $s_{x,e}(t_i)$, the fact still remains that clause (7) only specifies what to do for certain pairs (x_1, t_1), rather than for all such pairs. Of course it is possible to complicate the specification of $s_{x,e}(\lambda x_1.t_1)$ by saying what to do when x_1 does occur freely in e and arrive at a construction for $s_{x,e}$ (either by giving up structural properties and using a less natural recursion on the height of trees; or by using structural recursion to define a more general operation of simultaneous substitution [21]). An alternative approach, and one that works with the original simple specification, is to construct functions by giving rule-based inductive definitions of their graphs (with the rules encoding the required properties of the function); one then has to prove (using rule-based induction) that the resulting relations are single-valued, total and respect $=_\alpha$. This is in principle a fully formal and widely applicable approach to constructing functions like $s_{x,e}$ (using tools that in any case are part and parcel of structural operational semantics), but one that is extremely tedious to carry out. It would be highly preferable to establish a recursion principle that goes straight from definitions like (1)–(3) to the existence of the function $(x := t)(-) \in \Lambda/{=_\alpha} \to \Lambda/{=_\alpha}$. We provide such a principle here for a general class of signatures in which binding information can be declared. We call it *α-structural recursion* and it comes with a derived induction principle, *α-structural induction*.

These recursion and induction principles for α-equivalence classes of ASTs are simplifications and generalisations of the ones introduced by Gabbay and the author in [8] as part of a new mathematical model of fresh names and name binding. That paper expresses its results in terms of a non-standard axiomatic set theory, based on the classical Fraenkel-Mostowski permutation model of set theory. Experience shows that this formalism impedes the take up within computer science of the new ideas contained in [8]. There is an essentially equivalent, but more concrete description of the model as standard sets equipped with some simple extra structure. These so-called *nominal sets* are introduced in [16] and I will use them here to express α-structural recursion and induction within "ordinary mathematics", or more precisely, within Church's higher-order logic [3].

2 Nominal Syntax

The usual principles of structural recursion and induction are parameterised by an algebraic signature that specifies the allowed constructors for forming ASTs of each sort. In order to state principles of recursion and induction for α-equivalence classes of ASTs, we need to fix a notion of signature that also specifies the forms of binding that occur in the ASTs. As explained in the Introduction, we stick with the usual "nominal" approach in which bound entities are explicitly named. Any generalisation of the notion of algebraic signature to encompass constructors that bind names needs to specify how bound occurrences of names in an AST are associated with a binding site further up the syntax tree. There are a number of such mechanisms in the literature of varying degrees of generality [10, 17, 5, 12, 22]. Here we will use the notion of *nominal signature* from [22]. It has the advantage of dealing with binding and α-equivalence independently of any considerations to do with variables, substitution and β-equivalence; bound names in a nominal signature may be of several different sorts and not just variables that can be substituted for. In common with the other cited approaches, nominal signatures only allow for constructors that bind a fixed number of names (and without loss of much generality, we can take that number to be one). There are certainly forms of binding occurring "in the wild" that do not fit comfortably into this framework.[2] I believe that the notion of α-structural recursion given here can be extended to cover more general forms of statically scoped binding; but for simplicity's sake I will stick with constructors binding a fixed number of names.

2.1 Atoms, Nominal Signatures and Terms

From a logical point of view[3], the names we use for making localised bindings in formal languages only need to be atomic, in the sense that the structure of names (of the same kind) is immaterial compared with the distinctions between names. Therefore we will use the term *atom* for such names. Throughout this paper we fix two sets: the set $\mathbb{A}$ of all **atoms** and the set $\mathbb{AS}$ of all **atom-sorts**. We also fix a function $sort \in \mathbb{A} \to \mathbb{AS}$ assigning sorts to atoms and assume that the sets $\mathbb{AS}$ and $\mathbb{A}_{\mathsf{a}} \triangleq \{a \in \mathbb{A} \mid sort(a) = \mathsf{a}\}$, for each $\mathsf{a} \in \mathbb{AS}$, are all countably infinite.

A **nominal signature** Σ consists of a subset $\Sigma_{\mathrm{A}} \subseteq \mathbb{AS}$ of atom-sorts, a set Σ_{D} of **data-sorts** and a set Σ_{C} of **constructors**. Each constructor $K \in \Sigma_{\mathrm{C}}$ comes with an **arity** σ and a **result sort** $\mathsf{s} \in \Sigma_{\mathrm{D}}$, and we write $K : \sigma \to \mathsf{s}$ to indicate this information. The arities σ of Σ are given as follows; at the same time we define the **terms**[4] t over Σ of each arity, writing $t : \sigma$ to indicate that t has arity σ.

Atoms: every atom-sort $\mathsf{a} \in \Sigma_{\mathrm{A}}$ is an arity. If $a \in \mathbb{A}_{\mathsf{a}}$ is an atom of sort a, then $a : \mathsf{a}$.
Constructed terms: every data-sort $\mathsf{s} \in \Sigma_{\mathrm{D}}$ is an arity. If $K : \sigma \to \mathsf{s}$ is in Σ_{C} and $t : \sigma$, then $K\,t : \mathsf{s}$.

[2] The full version of $F_{<:}$ with records and pattern-matching used in Part 2B of the "POPLMARK challenge" (`www.cis.upenn.edu/group/proj/plclub/mmm/`) is an example.

[3] As opposed to a pragmatic one that also encompasses issues of parsing and pretty-printing.

[4] Compared with [22, Definition 2.3] we only define *ground* terms, since we do not need to consider variables ranging over terms here.

Tuples: if $\sigma_1, \ldots, \sigma_n$ is a finite list of arities, then $\sigma_1 * \cdots * \sigma_n$ is an arity; the $n = 0$ case is called the **unit** arity and written 1. If $t_1 : \sigma_1, \ldots, t_n : \sigma_n$, then $\langle t_1, \ldots, t_n \rangle : \sigma_1 * \cdots * \sigma_n$; in particular, when $n = 0$ we have $\langle\rangle : 1$.

Atom-binding: if $\mathsf{a} \in \Sigma_{\mathrm{A}}$ and σ is an arity, then $\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma$ is an arity. If $a \in \mathbb{A}_{\mathsf{a}}$ and $t : \sigma$, then $\langle\!\langle a \rangle\!\rangle t : \langle\!\langle \mathsf{a} \rangle\!\rangle \sigma$.

We write $Ar(\Sigma)$ for the set of all arities over a nominal signature Σ, $\mathsf{T}(\Sigma)$ for the set of all terms over Σ, and $ar \in \mathsf{T}(\Sigma) \to Ar(\Sigma)$ for the function assigning to each term t the unique arity σ such that $t : \sigma$ holds. For each $\sigma \in Ar(\Sigma)$, we write $\mathsf{T}(\Sigma)_\sigma$ for the subset $\{t \in \mathsf{T}(\Sigma) \mid ar(t) = \sigma\}$ of terms of arity σ.

Example 1. Here is a nominal signature for the version of the Milner-Parrow-Walker π-calculus given in [19, Definition 1.1.1]. (The sort gsum is for processes that are guarded sums, and the sort pre is for prefixed processes.)

atom-sorts	data-sorts	constructors
chan	proc	$Gsum : \mathsf{gsum} \to \mathsf{proc}$
	gsum	$Par : \mathsf{proc} * \mathsf{proc} \to \mathsf{proc}$
	pre	$Res : \langle\!\langle \mathsf{chan} \rangle\!\rangle \mathsf{proc} \to \mathsf{proc}$
		$Rep : \mathsf{proc} \to \mathsf{proc}$
		$Zero : 1 \to \mathsf{gsum}$
		$Pre : \mathsf{pre} \to \mathsf{gsum}$
		$Plus : \mathsf{gsum} * \mathsf{gsum} \to \mathsf{gsum}$
		$Out : \mathsf{chan} * \mathsf{chan} * \mathsf{proc} \to \mathsf{pre}$
		$In : \mathsf{chan} * \langle\!\langle \mathsf{chan} \rangle\!\rangle \mathsf{proc} \to \mathsf{pre}$
		$Tau : 1 \to \mathsf{pre}$
		$Match : \mathsf{chan} * \mathsf{chan} * \mathsf{pre} \to \mathsf{pre}$

This example uses several atom- and data-sorts, but does not illustrate the usefulness of the inter-mixing of the arity-formers for tupling and atom-binding that is allowed in a nominal signature. An example that does do this is given in [22, Example 2.2]; see also the discussion in [10, Sect. 3].

2.2 Ordinary Structural Recursion and Induction

The terms over a nominal signature Σ are just the abstract syntax trees determined by an ordinary signature associated with Σ whose sorts are the arities of Σ, whose constructors are those of Σ, plus constructors for tupling and atom-binding, and with atoms regarded as particular constants. Consequently we can use ordinary structural recursion to define functions on the set $\mathsf{T}(\Sigma)$ of terms over Σ. We state without proof a simple, iterative form of the principle that we will be using later.

Theorem 2. *Let Σ be a nominal signature. Suppose we are given sets S_σ, for each $\sigma \in Ar(\Sigma)$, and functions*

$$
\begin{aligned}
g_{\mathsf{a}} &\in \mathbb{A}_{\mathsf{a}} \to S_{\mathsf{a}} && (\mathsf{a} \in \Sigma_{\mathrm{A}}) \\
g_K &\in S_\sigma \to S_{\mathsf{s}} && ((K : \sigma \to \mathsf{s}) \in \Sigma_{\mathrm{C}}) \\
g_{\sigma_1 * \cdots * \sigma_n} &\in S_{\sigma_1} \times \cdots \times S_{\sigma_n} \to S_{\sigma_1 * \cdots * \sigma_n} && (\sigma_i \in Ar(\Sigma) \mid i = 1..n) \\
g_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma} &\in \mathbb{A}_{\mathsf{a}} \times S_\sigma \to S_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma} && (\mathsf{a} \in \Sigma_{\mathrm{A}}, \sigma \in Ar(\Sigma)).
\end{aligned}
$$

Then there is a unique family of functions $\overline{g}_\sigma \in \mathsf{T}(\Sigma)_\sigma \to S_\sigma$ *(*$\sigma \in Ar(\Sigma)$*) satisfying the following properties*

$$\overline{g}\, a = g_{\mathsf{a}}(a) \tag{8}$$
$$\overline{g}(K\, t) = g_K(\overline{g}\, t) \tag{9}$$
$$\overline{g}\langle t_1, \ldots, t_n\rangle = g_{\sigma_1 * \cdots * \sigma_n}\langle \overline{g}\, t_1, \ldots, \overline{g}\, t_n\rangle \tag{10}$$
$$\overline{g}\langle\!\langle a\rangle\!\rangle t = g_{\langle\!\langle \mathsf{a}\rangle\!\rangle \sigma}(a, \overline{g}\, t) \tag{11}$$

where we have abbreviated $\overline{g}_\sigma(t)$ *to* $\overline{g}\, t$ *(since* $\sigma = ar(t)$ *is determined by* t*).* □

Using the fact that subsets of $\mathsf{T}(\Sigma)$ are in bijection with functions $\mathsf{T}(\Sigma) \to \mathbb{B}$ (where $\mathbb{B} = \{T, F\}$ is the two-element set of boolean values), one can derive the following principle of structural induction for terms over Σ as a corollary of the uniqueness part of Theorem 2.

Corollary 3. *Let* Σ *be a nominal signature and* $S \subseteq \mathsf{T}(\Sigma)$ *a set of terms over* Σ*. To prove that* S *is the whole of* $\mathsf{T}(\Sigma)$ *it suffices to show*

$$(\forall \mathsf{a} \in \Sigma_{\mathrm{A}}, a \in \mathbb{A}_{\mathsf{a}})\ a \in S$$
$$(\forall (K : \sigma \to \mathsf{s}) \in \Sigma_{\mathrm{C}}, t : \sigma)\ t \in S \ \Rightarrow\ K\, t \in S$$
$$(\forall (\sigma_i \in Ar(\Sigma), t_i : \sigma_i \mid i = 1..n))\ t_1 \in S\ \&\ \cdots\ \&\ t_n \in S \ \Rightarrow\ \langle t_1, \ldots, t_n\rangle \in S$$
$$(\forall \mathsf{a} \in \Sigma_{\mathrm{A}}, a \in \mathbb{A}_{\mathsf{a}}, \sigma \in Ar(\Sigma), t : \sigma)\ t \in S \ \Rightarrow\ \langle\!\langle a\rangle\!\rangle t \in S\ .$$

□

2.3 α-Equivalence and α-Terms

So far we have taken no account of the fact that atom-binder terms $\langle\!\langle a\rangle\!\rangle t$ should be identified up to renaming the bound atom a. Given a nominal signature Σ, the binary relation of **α-equivalence**, $t =_\alpha t' : \sigma$ (where $\sigma \in Ar(\Sigma)$ and $t, t' \in \mathsf{T}(\Sigma)_\sigma$) makes such identifications. It is inductively defined by the following rules. In rule ($=_\alpha$-4), $atm\, t$ indicates the finite set of atoms occurring in t; and $t\{a'/a\}$ indicates the term resulting from replacing all occurrences in t of the atom a by the atom a' (assumed to be of the same sort).

$$(=_\alpha\text{-}1)\ \frac{\mathsf{a} \in \Sigma_{\mathrm{A}} \qquad a \in \mathbb{A}_{\mathsf{a}}}{a =_\alpha a : \mathsf{a}} \qquad (=_\alpha\text{-}2)\ \frac{(K : \sigma \to \mathsf{s}) \in \Sigma_{\mathrm{C}} \qquad t =_\alpha t' : \sigma}{K\, t =_\alpha K\, t' : \mathsf{s}}$$

$$(=_\alpha\text{-}3)\ \frac{t_1 =_\alpha t_1' : \sigma_1 \qquad \cdots \qquad t_n =_\alpha t_n' : \sigma_n}{\langle t_1, \ldots, t_n\rangle =_\alpha \langle t_1', \ldots, t_n'\rangle : \sigma_1 * \cdots * \sigma_n}$$

$$(=_\alpha\text{-}4)\ \frac{\mathsf{a} \in \Sigma_{\mathrm{A}} \qquad a, a', a'' \in \mathbb{A}_{\mathsf{a}} \qquad a'' \notin atm\,\langle a, t, a', t'\rangle \qquad t\{a''/a\} =_\alpha t'\{a''/a'\} : \sigma}{\langle\!\langle a\rangle\!\rangle t =_\alpha \langle\!\langle a'\rangle\!\rangle t' : \langle\!\langle \mathsf{a}\rangle\!\rangle \sigma}$$

Here we have generalised to terms over a nominal signature a version of the definition of α-equivalence of λ-terms [11, p. 36] that is conveniently syntax-directed compared with the classic version [1, Definition 2.1.11]. It is easy to see that $=_\alpha$ is reflexive, symmetric and respects the various term-forming constructions for nominal syntax. Less straightforward is the fact that $=_\alpha$ is transitive. This can be proved in a number of ways. My favourite way makes good use of the techniques we will be using in Sect. 3, based on the action of atom-permutations on terms; see [16, Example 1].

For each $\sigma \in Ar(\Sigma)$, we write $\mathsf{T}_\alpha(\Sigma)_\sigma$ for the quotient of $\mathsf{T}(\Sigma)_\sigma$ by the equivalence relation $(-) =_\alpha (-) : \sigma$. Thus the elements of $\mathsf{T}_\alpha(\Sigma)_\sigma$ are α-equivalence classes of terms of arity σ; we write $[t]_\alpha$ for the class of t and refer to $[t]_\alpha$ as an **α-term** of arity σ over the nominal signature Σ.

3 Finite Support

The crucial ingredient in the formulation of structural recursion and induction for α-terms over a nominal signature is the notion of finite[5] *support*. It gives a well-behaved way, phrased in terms of atom-permutations, of expressing the fact that atoms are fresh for mathematical objects. It turns out to agree with the obvious definition when the objects are finite data such as abstract syntax trees, but allows us to deal with freshness for the not so obvious case of infinite sets and functions.

3.1 Nominal Sets

Let $Perm$ denote the set of all (finite, sort-respecting) **atom-permutations**; by definition, its elements are bijections $\pi : \mathbb{A} \leftrightarrow \mathbb{A}$ such that $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is finite and $sort(\pi(a)) = sort(a)$ for all $a \in \mathbb{A}$. The operation of composing bijections gives a binary operation $\pi, \pi' \in Perm \mapsto \pi \circ \pi' \in Perm$ that makes $Perm$ into a group; we write ι for the identity atom-permutation and π^{-1} for the inverse of π. Among the elements of $Perm$ we single out **transpositions** $(a\ a')$ given by a pair of atoms of the same sort; $(a\ a')$ is the atom-permutation mapping a to a', mapping a' to a and leaving all other atoms fixed. It is a basic fact of group theory that every $\pi \in Perm$ is equal to a finite composition of such transpositions.

An **action** of $Perm$ on a set X is a function $Perm \times X \to X$, whose effect on $(\pi, x) \in Perm \times X$ we write as $\pi \cdot x$ (with X understood), and which is required to have the properties: $\iota \cdot x = x$ and $\pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x$, for all $x \in X$ and $\pi, \pi' \in Perm$. Given such an action and an element $x \in X$, we say that a set $A \subseteq \mathbb{A}$ of atoms **supports** x if $(a\ a') \cdot x = x$ holds for all atoms a and a' (of the same sort) that are *not* in A. Then a **nominal set** is by definition a set X equipped with an action of $Perm$ such that every element $x \in X$ is supported by some *finite* set of atoms. If A_1 and A_2 are both finite sets of atoms supporting $x \in X$, then one can show that $A_1 \cap A_2$ also supports x. It follows that in a nominal set X, each element $x \in X$ possesses a *smallest* finite support, which we write as $supp_X(x)$ (or just $supp(x)$, if X is clear from the context) and call **the support of** x in X.

[5] Both Gabbay [7] and Cheney [2] develop more general notions of "small" supports. As Cheney's work shows, such a generalisation is necessary for some techniques of classical model theory to be applied; but finite supports are sufficient here.

Example 4. (i) Each set $\mathbb{A}_{\mathsf{a}}$ of atoms of a particular sort a is a nominal set once we endow it with the atom-permutation action given by $\pi \cdot a = \pi(a)$; as one might expect, $supp(a) = \{a\}$. It is not hard to see that the disjoint union of nominal sets is again a nominal set. So since the set of all atoms is the disjoint union of $\mathbb{A}_{\mathsf{a}}$ as a ranges over atom-sorts, $\mathbb{A}$ is a nominal set with atom-permutation action and support sets as for each individual $\mathbb{A}_{\mathsf{a}}$.

(ii) Let Σ be a nominal signature. Using Theorem 2 we can define an atom-permutation action on the sets $\mathsf{T}(\Sigma)_\sigma$ of terms over Σ of each arity $\sigma \in Ar(\Sigma)$:

$$\begin{array}{cc} \pi \cdot a = \pi(a) & \pi \cdot \langle t_1, \ldots, t_n \rangle = \langle \pi \cdot t_1, \ldots, \pi \cdot t_n \rangle \\ \pi \cdot K\, t = K(\pi \cdot t) & \pi \cdot \langle\!\langle a \rangle\!\rangle t = \langle\!\langle \pi \cdot a \rangle\!\rangle (\pi \cdot t)\ . \end{array}$$

Using Corollary 3 one can prove that this has the properties required of an atom-permutation action, that $a, a' \notin atm\, t \Rightarrow (a\ a') \cdot t = t$, and that $a \in atm\, t\ \&\ (a\ a') \cdot t = t \Rightarrow a = a'$. From these facts it follows that each $\mathsf{T}(\Sigma)_\sigma$ is a nominal set, with $supp(t) = atm\, t$, the finite set of atoms occurring in t.

(iii) Turning next to α-terms over Σ (Sect. 2.3), first note that the action of atom-permutations on terms preserves α-equivalence.[6] Therefore we get a well-defined action on α-terms by defining: $\pi \cdot [t]_\alpha = [\pi \cdot t]_\alpha$. For this action one finds that $\mathsf{T}_\alpha(\Sigma)_\sigma$ is a nominal set with $supp([t]_\alpha) = fa(t)$, the finite set of **free atoms** of any representative t of the class $[t]_\alpha$, defined (using Theorem 2) by:

$$\begin{array}{cc} fa(a) = \{a\} & fa(\langle t_1, \ldots, t_n \rangle) = fa(t_1) \cup \cdots \cup fa(t_n) \\ fa(K\, t) = fa(t) & fa(\langle\!\langle a \rangle\!\rangle t) = fa(t) - \{a\}\ . \end{array}$$

(iv) Each set S becomes a nominal set, called the **discrete nominal set** on S, if we endow it with the **trivial action** of atom-permutations, given by $\pi \cdot s = s$ for each $\pi \in Perm$ and $s \in S$; in this case the support of each element is empty. In particular, we will regard the set of booleans $\mathbb{B} = \{T, F\}$ and the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ as nominal sets in this way.

3.2 Products and Functions

If $X_1, \ldots, X_n$ are nominal sets, then we get an action of atom-permutations on their cartesian product $X_1 \times \cdots \times X_n$ by defining $\pi \cdot (x_1, \ldots, x_n)$ to be $(\pi \cdot x_1, \ldots, \pi \cdot x_n)$, for each $(x_1, \ldots, x_n) \in X_1 \times \cdots \times X_n$. If A_i supports $x_i \in X_i$ for $i = 1..n$, then it is not hard to see that $A_1 \cup \cdots \cup A_n$ supports $(x_1, \ldots, x_n) \in X_1 \times \cdots \times X_n$; indeed, one can prove that $supp((x_1, \ldots, x_n)) = supp(x_1) \cup \cdots \cup supp(x_n)$. Thus $X_1 \times \cdots \times X_n$ is also a nominal set.

If X and Y are nominal sets, then we get an action of atom-permutations on the set $X \to Y$ of all functions from X to Y by defining $\pi \cdot f$ to be the function mapping each $x \in X$ to $\pi \cdot (f(\pi^{-1} \cdot x)) \in Y$. If you have not seen this definition before, it may look

[6] As noted in [16, p 169], this fact has nothing much to do with the particular nature of $=_\alpha$ and everything to do with the fact that it is inductively defined by a collection of schematic rules with the property that the action of any atom-permutation takes any instance of the rules to another instance.

more complicated than expected; however, it is forced by the important requirement that function application be respected by atom-permutations:[7]

$$\pi \cdot (f(x)) = (\pi \cdot f)(\pi \cdot x)\ . \tag{12}$$

Unlike the situation for cartesian product, not every element $f \in X \to Y$ is necessarily finitely supported with respect to this action (see Example 6 below). However, note that if f is supported by a finite set of atoms A, then $\pi \cdot f$ is supported by $\{\pi(a) \mid a \in A\}$. Therefore

$$X \to_{\mathrm{fs}} Y \triangleq \{f \in X \to Y \mid (\exists \text{finite } A \subseteq \mathbb{A})\ A \text{ supports } f\}$$

is closed under the atom-permutation action and is a nominal set.

Example 5. Recall that the elements of $Perm$ are bijections from $\mathbb{A}$ to itself that respect sorts and leave fixed all but finitely many atoms. So each $\pi \in Perm$ is in particular a function $\mathbb{A} \to \mathbb{A}$. Regarding $\mathbb{A}$ as a nominal set as in Example 4(i), the action of atom-permutations on π *qua* function turns out to be the operation of *conjugation*: $\pi' \cdot \pi = \pi' \circ \pi \circ (\pi')^{-1}$. Hence the action of atom-permutations on $\mathbb{A} \to \mathbb{A}$ restricts to an action on $Perm$. One can prove that the finite set $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ supports π with respect to this action (and is in fact the smallest such set); so $Perm$ is a nominal set.

Example 6. Not every function between nominal sets is finitely supported. For example, since the set $\mathbb{A}$ of atoms is countable, there are surjective functions in $\mathbb{N} \to \mathbb{A}$; but it is not hard to see that any $f \in \mathbb{N} \to_{\mathrm{fs}} A$ must have a finite image (which is in fact the support of f). A more subtle example of a non-finitely-supported function is any **choice function** for the set $\mathbb{A}$ of atoms, i.e. any function $choose \in (\mathbb{A} \to_{\mathrm{fs}} \mathbb{B}) \to \mathbb{A}$ (where $\mathbb{B} = \{T, F\}$) satisfying $f(a) = T \ \Rightarrow\ f(choose(f)) = T$, for all $f \in \mathbb{A} \to_{\mathrm{fs}} \mathbb{B}$ and $a \in \mathbb{A}$.[8]

3.3 Freshness

Given an element of a nominal set, most of the time we are interested not so much in its support as in the (infinite) set of atoms that are *not* in its support. More generally, if $x \in X$ and $y \in Y$ are elements of nominal sets, we write $x \# y$ when $supp_X(x) \cap supp_Y(y) = \emptyset$ and say that x is **fresh** for y. Of the many properties of this notion of "freshness" developed in [8, 16] we single out the following one that we need below; it provides a very general criterion for when a construction that "picks a fresh atom" is independent of which fresh atom is chosen. (We omit the proof in this abstract.)

Lemma 7 (Freshness Lemma). *Given an atom-sort* $\mathsf{a} \in \mathbb{AS}$ *and a nominal set* X*, if a finitely supported function* $h \in \mathbb{A}_{\mathsf{a}} \to_{\mathrm{fs}} X$ *satisfies*

[7] More precisely, the definition of the action on functions is forced by the requirement that $X \to Y$ together with the usual application function be the *exponential* of X and Y in the cartesian closed category of sets equipped with an atom-permutation action.

[8] It was this lack of finite support for choice functions that motivated the original construction of the permutation model of set theory by Fraenkel and Mostowski.

$$(\exists a \in \mathbb{A}_{\mathsf{a}})\ a \# h \ \ \&\ \ a \# h(a) \tag{13}$$

then there is a unique element $\mathit{fresh}(h) \in X$ *satisfying*

$$(\forall a \in \mathbb{A}_{\mathsf{a}})\ a \# h \ \Rightarrow\ h(a) = \mathit{fresh}(h)\ . \tag{14}$$

□

4 Recursion and Induction for α-Terms

4.1 The Structure of α-Terms

Recall that $\mathsf{T}_\alpha(\Sigma)_\sigma$ denotes the set of α-terms of arity σ over a nominal signature Σ; by definition these are α-equivalence classes $[t]_\alpha$ of terms $t : \sigma$. Elementary properties of the relation $=_\alpha$ of α-equivalence yield the following structural properties of α-terms; at the same time we introduce some concrete syntax for α-terms mirroring the informal notation for α-equivalence classes mentioned in the Introduction.

Atoms: if $\mathsf{a} \in \Sigma_{\mathrm{A}}$ and $e \in \mathsf{T}_\alpha(\Sigma)_{\mathsf{a}}$, then there is a unique $a \in \mathbb{A}_{\mathsf{a}}$ such that $e = [a]_\alpha$. *In this case we write* e *just as* a.

Constructed α-terms: if $\mathsf{s} \in \Sigma_{\mathrm{D}}$ and $e \in \mathsf{T}_\alpha(\Sigma)_{\mathsf{s}}$, then there are unique $(K : \sigma \to \mathsf{s}) \in \Sigma_{\mathrm{C}}$ and $e' \in \mathsf{T}_\alpha(\Sigma)_\sigma$ such that there exists t' with $e' = [t']_\alpha$ and $e = [K\,t']_\alpha$. *In this case we write* e *as* $K\,e'$.

Tuples: if $\sigma_1, \ldots, \sigma_n \in Ar(\Sigma)$ and $e \in \mathsf{T}_\alpha(\Sigma)_{\sigma_1 * \cdots * \sigma_n}$, then there are unique $e_i \in \mathsf{T}_\alpha(\Sigma)_{\sigma_i}$ for $i = 1..n$ such that there exist t_i with $e_i = [t_i]_\alpha$ $(i = 1..n)$ and $e = [\langle t_1, \ldots, t_n \rangle]_\alpha$. *In this case we write* e *as* $(e_1, \ldots, e_n)$.

Atom-binding: if $\mathsf{a} \in \Sigma_{\mathrm{A}}$, $\sigma \in Ar(\Sigma)$ and $e \in \mathsf{T}_\alpha(\Sigma)_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}$, then for each $a \in \mathbb{A}_{\mathsf{a}}$ with $a \# e$ (i.e. with a not a free atom of e—cf. Example 4(iii)), there is a unique $e' \in \mathsf{T}_\alpha(\Sigma)_\sigma$ such that there exists t' with $e' = [t']_\alpha$ and $e = [\langle\!\langle a \rangle\!\rangle t']_\alpha$. *In this case we write* e *as* $a.\,e'$.

4.2 α-Structural Recursion and Induction

We can now state and prove the main result of this paper, a principle of structural recursion for α-terms over a nominal signature. Compared with Theorem 2, the principle uses nominal sets rather than ordinary sets, and requires a common finite support for the collection of functions in its hypothesis. Furthermore, the function supplied for each binding arity must satisfy a *freshness condition for binders* (FCB) saying, roughly, that for *some* sufficiently fresh choice of the atom being bound, the result of the function can never contain that atom in its support. These conditions ensure that there is a unique (finitely supported) arity-indexed family of functions that is well-defined on α-equivalence classes and satisfies the required recursion equations—for *all* sufficiently fresh bound atoms, in the case of the recursion equation for binders. The "*some/any*" aspect of the principle is a characteristic of the treatment of fresh names from [8].

Theorem 8 (α-Structural recursion). *Let* Σ *be a nominal signature. Suppose we are given an arity-indexed family of nominal sets* X_σ $(\sigma \in Ar(\Sigma))$ *and functions*

$$
\begin{array}{rl}
f_{\mathsf{a}} \in \mathbb{A}_{\mathsf{a}} \to_{\mathrm{fs}} X_{\mathsf{a}} & (\mathsf{a} \in \Sigma_{\mathrm{A}}) \\
f_K \in X_\sigma \to_{\mathrm{fs}} X_{\mathsf{s}} & ((K : \sigma \to_{\mathrm{fs}} \mathsf{s}) \in \Sigma_{\mathrm{C}}) \\
f_{\sigma_1 * \cdots * \sigma_n} \in X_{\sigma_1} \times \cdots \times X_{\sigma_n} \to_{\mathrm{fs}} X_{\sigma_1 * \cdots * \sigma_n} & (\sigma_i \in Ar(\Sigma) \mid i = 1..n) \\
f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma} \in \mathbb{A}_{\mathsf{a}} \times X_\sigma \to_{\mathrm{fs}} X_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma} & (\mathsf{a} \in \Sigma_{\mathrm{A}}, \sigma \in Ar(\Sigma))
\end{array}
$$

all of which are supported by a finite set of atoms A and satisfy the

Freshness Condition for Binders (FCB): *for each atom-binding arity $\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma \in Ar(\Sigma)$, the function $f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}$ satisfies $(\exists a' \in \mathbb{A}_{\mathsf{a}} - A)(\forall x \in X_\sigma)\, a' \# f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}(a', x)$.*

Then there is a unique family of finitely supported functions $\overline{f}_\sigma \in \mathsf{T}_\alpha(\Sigma)_\sigma \to_{\mathrm{fs}} X_\sigma$ ($\sigma \in Ar(\Sigma)$) with $supp(\overline{f}_\sigma) \subseteq A$ and satisfying the following properties for all $a, e, e_1, \ldots, e_n$ of suitable arity:

$$\overline{f} a = f_{\mathsf{a}}(a) \tag{15}$$

$$\overline{f}(K\, e) = f_K(\overline{f} e) \tag{16}$$

$$\overline{f}(e_1, \ldots, e_n) = f_{\sigma_1 * \cdots * \sigma_n}(\overline{f} e_1, \ldots, \overline{f} e_n) \tag{17}$$

$$a \notin A \;\Rightarrow\; \overline{f}(a.\, e) = f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}(a, \overline{f} e) \tag{18}$$

where we have abbreviated $\overline{f}_\sigma(e)$ to $\overline{f} e$ and used the notation for α-terms from Sect. 4.1.

Proof (sketch). We can reduce the proof of the theorem to an application of Theorem 2, taking advantage of the fact that we are working (informally) in higher-order logic.[9] From the $Ar(\Sigma)$-indexed family of nominal sets X_σ we define another such family: $S_\sigma \triangleq Perm \to_{\mathrm{fs}} X_\sigma$ (regarding $Perm$ as a nominal set as in Example 5 and using the $\to_{\mathrm{fs}}$ construct from Sect.3.2). Now define functions g_{a}, g_K, $g_{\sigma_1 * \cdots * \sigma_n}$ and $g_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}$ (with domains and codomains as in the statement of Theorem 2) as follows.

$$
\begin{aligned}
g_{\mathsf{a}}\, a &\triangleq \lambda \pi \in Perm.\ f_{\mathsf{a}}(\pi(a)) \\
g_K\, s &\triangleq \lambda \pi \in Perm.\ f_K(s(\pi)) \\
g_{\sigma_1 * \cdots * \sigma_n}(s_1, \ldots, s_n) &\triangleq \lambda \pi \in Perm.\ f_{\sigma_1 * \cdots * \sigma_n}(s_1(\pi), \ldots, s_n(\pi)) \\
g_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}(a, s) &\triangleq \lambda \pi \in Perm.\ fresh(\lambda a' \in \mathbb{A}_{\mathsf{a}}.\, f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}(a', s(\pi \circ (a\ a'))))
\end{aligned}
$$

The crucial clause in this definition is the last one, where we are using the *fresh* functional from Lemma 7 applied to the function $h \triangleq \lambda a' \in \mathbb{A}_{\mathsf{a}}.\, f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}(a', s(\pi \circ (a\ a')))$; in this abstract we omit the proof that the condition (13) needed to apply the lemma is satisfied in this case. Applying Theorem 2 with this data, we get a family of functions $\overline{g}_\sigma \in \mathsf{T}(\Sigma)_\sigma \to (Perm \to_{\mathrm{fs}} X_\sigma)$ satisfying the recursion equations (8)–(11) of that theorem. Next one proves that these functions respect α-equivalence; this is done by induction over the derivation of $t_1 =_\alpha t_2 : \sigma$ from the rules in Sect.2.3. So the functions $\overline{g}_\sigma$ induce functions $\overline{f}_\sigma \in \mathsf{T}_\alpha(\Sigma)_\sigma \to X_\sigma$ given by $\overline{f}_\sigma[t]_\alpha \triangleq \overline{g}_\sigma\, t\, \iota$ for any $t : \sigma$ (recalling that ι stands for the identity permutation). One proves that these functions $\overline{f}_\sigma$ are all supported by A by first proving that the functions $\overline{g}_\sigma$ are so supported. Then

[9] In other words the theorem is reducible to primitive recursion at higher types.

the fact that the $\overline{f}_\sigma$ satisfy the required recursion equations (15)–(18) follows from the recursion equations (8)–(11) satisfied by the $\overline{g}_\sigma$. That concludes the existence part of the proof of Theorem 8.

For the uniqueness part, suppose functions $f'_\sigma \in \mathsf{T}_\alpha(\Sigma)_\sigma \to_{\mathrm{fs}} X_\sigma$ are all supported by A and satisfy the recursion equations (15)–(18) for $\overline{f}_\sigma$. Define $g'_\sigma \in \mathsf{T}(\Sigma)_\sigma \to S_\sigma$ by $g'_\sigma\, t\, \pi \triangleq f'_\sigma[\pi \cdot t]_\alpha$ ($\sigma \in Ar(\Sigma), t : \sigma, \pi \in Perm$). One can show that the g'_σ satisfy the same recursion equations (8)–(11) from Theorem 2 as the functions $\overline{g}_\sigma$; so by the uniqueness part of that theorem, $g'_\sigma = \overline{g}_\sigma$. Therefore for all $t : \sigma$, $f'_\sigma[t]_\alpha = f'_\sigma[\iota \cdot t]_\alpha \triangleq g'_\sigma\, t\, \iota = \overline{g}_\sigma\, t\, \iota \triangleq \overline{f}_\sigma[t]_\alpha$; hence $f'_\sigma = \overline{f}_\sigma$. □

Remark 9 ("Some/any" property). In Theorem 8 we gave the FCB as an existential statement. It is in fact equivalent to the universal statement $(\forall a' \in \mathbb{A}_{\mathsf{a}} - A)(\forall x \in X_\sigma)\ a' \# f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}(a', x)$. This is an instance of the characteristic "some/any" property of fresh names noted in [8, Proposition 4.10] and [16, Proposition 4].

Given a nominal set X, we can use the usual bijection between subsets of X and functions in $X \to \mathbb{B}$ (where $\mathbb{B} = \{T, F\}$) to transfer the action of atom-permutations on $X \to \mathbb{B}$ to one on subsets of X. This action sends $\pi \in Perm$ and $S \subseteq X$ to the subset $\pi \cdot S = \{\pi \cdot x \mid x \in S\}$. The nominal set $P_{\mathrm{fs}}(X)$ of **finitely supported subsets** of the nominal set X consists of all those subsets $S \subseteq X$ that are finitely supported with respect to this action. Thus $P_{\mathrm{fs}}(X)$ is isomorphic to $X \to_{\mathrm{fs}} \mathbb{B}$. Using this isomorphism, as a corollary of the uniqueness part of Theorem 8 we obtain the following principle of structural induction for α-terms.

Corollary 10 (α-Structural induction). *Let Σ be a nominal signature. Suppose we are given a finitely supported set $S \in P_{\mathrm{fs}}(\mathsf{T}_\alpha(\Sigma))$ of α-terms over Σ. To prove that S is the whole of $\mathsf{T}_\alpha(\Sigma)$ it suffices to show*

$$
\begin{array}{l}
(\forall \mathsf{a} \in \Sigma_{\mathrm{A}}, a \in \mathbb{A}_{\mathsf{a}})\ a \in S \\
(\forall (K : \sigma \to \mathsf{s}) \in \Sigma_{\mathrm{C}}, e \in \mathsf{T}_\alpha(\Sigma)_\sigma)\ e \in S \ \Rightarrow\ K\, e \in S \\
(\forall (\sigma_i \in Ar(\Sigma), e_i \in \mathsf{T}_\alpha(\Sigma)_{\sigma_i} \mid i = 1..n))\ e_1 \in S \,\&\, \cdots \,\&\, e_n \in S \ \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (e_1, \ldots, e_n) \in S \\
(\forall \mathsf{a} \in \Sigma_{\mathrm{A}}, \sigma \in Ar(\Sigma))(\exists a \in \mathbb{A}_{\mathsf{a}} - \mathit{supp}(S))(\forall e \in \mathsf{T}_\alpha(\Sigma)_\sigma)\ e \in S \ \Rightarrow\ a.\, e \in S\ .
\end{array}
$$

□

Remark 11 (Primitive recursion). Theorem 8 gives a simple "iterative" form of structural induction for α-terms, rather than a more complicated "primitive recursive" form with recursion equations

$$
\begin{array}{ll}
\overline{f}a = f_{\mathsf{a}}(a) & \overline{f}(e_1, \ldots, e_n) = f_{\sigma_1 * \cdots * \sigma_n}(e_1, \ldots, e_n, \overline{f}e_1, \ldots, \overline{f}e_n) \\
\overline{f}(K\, e) = f_K(e, \overline{f}e) & \mathit{atm} \notin A \Rightarrow \overline{f}(a.\, e) = f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}(a, e, \overline{f}e).
\end{array}
$$

In fact this more general form can be deduced from the simple one given in the theorem.

4.3 "Sort-Directed" Recursion Principle

Theorem 8 is an "arity-directed" recursion principle for α-terms: one has to specify nominal sets X_σ for each arity σ, and give functions $f_{(\_)}$ for tuple and atom-binding

binding arities in addition to ones for atoms and constructors. It is possible to derive a "sort-directed" version of the principle in which one only has to give X_σ when σ is a data-sort, and only has to give the functions $f_{(\_)}$ for constructors; the FCB has to be replaced by a more complicated family of conditions, indexed by the argument arities of constructors. In this extended abstract I will not formulate this version of the principle for an arbitrary nominal signature, but instead just give it for the particular case of untyped λ-calculus, for which the FCB is quite simple to state.

Let Σ^λ be the nominal signature with a single atom-sort v (for variables), a single data-sort t (for λ-terms), and constructors $Var : \mathsf{v} \to \mathsf{t}$, $App : \mathsf{t} * \mathsf{t} \to \mathsf{t}$ and $Lam : \langle\!\langle \mathsf{v} \rangle\!\rangle \mathsf{t} \to \mathsf{t}$. Thus $\mathsf{T}(\Sigma^\lambda)_\mathsf{t}$ is the usual set Λ of abstract syntax trees of λ-terms and $\mathsf{T}_\alpha(\Sigma^\lambda)_\mathsf{t}$ is the quotient $\Lambda/{=_\alpha}$ of that by the usual notion of α-equivalence—in other words $\mathsf{T}_\alpha(\Sigma^\lambda)_\mathsf{t}$ is what is normally meant by the set of all (open or closed) untyped λ-terms. The following "sort-directed" version of α-structural recursion for Σ^λ can be deduced by suitably instantiating $X_{(\_)}$ and $f_{(\_)}$ in Theorem 8.

Corollary 12 (α-Structural recursion for λ-terms). *Given a nominal set X and functions $h_{Var} \in \mathbb{A}_\mathsf{v} \to_{\mathrm{fs}} X$, $h_{App} \in X \times X \to_{\mathrm{fs}} X$ and $h_{Lam} \in \mathbb{A}_\mathsf{v} \times X \to_{\mathrm{fs}} X$, all supported by a finite set of atoms A and with h_{Lam} satisfying*

$$(\exists a' \in \mathbb{A}_\mathsf{v} - A)(\forall x \in X)\ a' \# h_{Lam}(a', x) \qquad \text{(FCB}')$$

then there is a unique finitely supported function $\overline{h} \in \Lambda/{=_\alpha} \to_{\mathrm{fs}} X$ with $supp(\overline{h}) \subseteq A$ and satisfying

$$\overline{h}(\mathit{Var}\, a) = h_{Var}(a) \tag{19}$$

$$\overline{h}(\mathit{App}\,(e_1, e_2)) = h_{App}(\overline{h}\, e_1, \overline{h}\, e_2) \tag{20}$$

$$a \notin A \;\Rightarrow\; \overline{h}(\mathit{Lam}\, a.\, e) = h_{Lam}(a, \overline{h}\, e)\ . \tag{21}$$

□

4.4 Applying the Principles

How do we use Theorem 8 in practice? Suppose that some language of interest has been specified as the α-terms for a particular nominal signature. Suppose that we wish to define a function on those α-terms specified by an instance of the recursion scheme (15)–(18) and we have identified suitable functions f_a, f_K, $f_{\sigma_1 * \cdots * \sigma_n}$ and $f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}$. Then there are three tasks involved in applying the theorem to this data:

(I) Show that the sets X_σ that we are mapping into have the structure of nominal sets.
(II) Show that the functions f_a, f_K, $f_{\sigma_1 * \cdots * \sigma_n}$ and $f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}$ are all supported by a single finite set of atoms A.
(III) Show that the functions $f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma}$ for atom-binding arities satisfy the FCB.

It is possible to dispose of tasks (I) and (II) by applying a single metatheorem about the notion of support, based on the fact that nominal sets form a model of higher-order logic (without choice functions—see Example 6). In the author's opinion, the best way of explaining this model is to use *topos theory* (see [13], for example). Call a function $f \in X \to Y$ between two nominal sets **equivariant** if it is supported by the empty

set; in view of (12), this means that $\pi \cdot (f(x)) = f(\pi \cdot x)$, for all $\pi \in Perm$ and $x \in X$. Nominal sets and equivariant functions form a category that has the structure of a boolean topos with natural number object: products and exponentials are given by the operations $(-) \times (-)$ and $(-) \to_{\mathrm{fs}} (-)$ considered in Sect.3.2; the terminal object, subobject classifier and the natural number object are just the discrete nominal sets 1, $\mathbb{B}$ and $\mathbb{N}$ respectively (cf. Example 4(iv)). As for any such category, there is a sound interpretation of classical higher-order logic with arithmetic in this category. However, in this particular case the interpretation is easy to describe concretely: so long as we interpret function variables as ranging over only finitely supported functions, the usual set-theoretic interpretation of higher-order logic always yields finitely supported elements. If we remain within pure higher-order logic over ground types for numbers and booleans, then we only get elements with empty support. However, if we add a ground type for the set $\mathbb{A}$ of atoms, a constant for the function $sort \in \mathbb{A} \to \mathbb{AS}$ (taking $\mathbb{AS}$ to be a copy of $\mathbb{N}$) and constants for each atom, then the terms and formulas of higher-order logic describe functions and subsets which may have non-empty, finite support; such a "higher-order logic with atoms" has been developed by Gabbay [7]. Note that nominal sets of abstract syntax trees $\mathsf{T}(\Sigma)$ and their quotients by α-equivalence $\mathsf{T}_\alpha(\Sigma)$ are constructible within such a setting. As far as tasks (I) and (II) are concerned, we can sum things up thus: *if we use nominal sets and finitely supported functions in constructions definable in classical higher-order logic with arithmetic but without choice, the result will again be nominal sets and finitely supported functions.*

5 Examples

Here are some examples of Theorem 8 and Corollary 12 in action. In view of the above remarks, in each case we pass quickly over tasks (I) and (II) and concentrate on task (III).

Example 13 (Capture-avoiding substitution). The example mentioned in the Introduction of capture-avoiding substitution of λ-terms, $\hat{s}_{x,e} \in \Lambda/{=_\alpha} \to \Lambda/{=_\alpha}$, is obtained from Corollary 12 (using the nominal signature Σ^λ) by taking X to be the nominal set $\Lambda/{=_\alpha}$, i.e. $\mathsf{T}_\alpha(\Sigma^\lambda)_{\mathsf{t}}$. Given $x \in \mathbb{A}_{\mathsf{v}}$ and $e \in X$, then $\hat{s}_{x,e}$ is given by $\overline{h}$ where

$$h_{Var} \triangleq \lambda a \in \mathbb{A}_{\mathsf{v}}.\ \mathit{if}\ a = x\ \mathit{then}\ e\ \mathit{else}\ \mathit{Var}\, a \qquad h_{Lam} \triangleq \lambda (a, e) \in \mathbb{A}_{\mathsf{v}} \times X.\ \mathit{Lam}\, a.\, e$$
$$h_{App} \triangleq \lambda (e_1, e_2) \in X \times X.\ \mathit{App}(e_1, e_2) \qquad A \triangleq \mathit{supp}(x, e).$$

(FCB′) is satisfied because, as noted in Example 4(iii), for each $e \in X = \mathsf{T}_\alpha(\Sigma)_{\mathsf{t}}$, $\mathit{supp}(e)$ is the finite set of free atoms of e; in particular $a \# \mathit{Lam}\, a.\, e = h_{Lam}(a, e)$, because a is not free in (any representative of the α-equivalence class) $\mathit{Lam}\, a.\, e$. Note that the common finite support A of the $h_{(\_)}$ functions consists of x and the finite set of free variables of e. Therefore the restriction "$a \notin A$" in the recursion equation (21) corresponds precisely to the side-condition "$x_1 \neq x$ and x_1 is not free in e" in (7).

Example 14 (Length of an α-term). In [9, Sect. 3.3] Gordon and Melham give the usual recursion scheme for defining the length of a λ-term, remark that it is not a direct instance of the scheme developed in that paper (their Axiom 4) and embark on a detour

via simultaneous substitutions to define the length function. This difficulty is analysed by Norrish [14, Sect. 3] on the way to his improved version of Gordon and Melham's recursion scheme (discussed further in Sect. 6). Pleasingly, the usual recursive definition of the length of a λ-term, or more generally of an α-term over any nominal signature, is a very simple application of α-structural recursion.[10] Thus in Theorem 8 we take X_σ to be the discrete nominal set $\mathbb{N}$ of natural numbers and

$$f_{\mathsf{a}} \triangleq \lambda a \in \mathbb{A}_{\mathsf{a}}.\ 1 \qquad f_{\sigma_1 * \cdots * \sigma_n} \triangleq \lambda(k_1, \ldots, k_n) \in \mathbb{N}^n.\ k_1 + \cdots + k_n$$
$$f_K \triangleq \lambda k \in \mathbb{N}.\ k + 1 \qquad f_{\langle\!\langle \mathsf{a} \rangle\!\rangle \sigma} \triangleq \lambda(a, k) \in \mathbb{A}_{\mathsf{a}} \times \mathbb{N}.\ k + 1$$

These functions are all supported by $A = \emptyset$ and the FCB holds trivially, because $a \# k$ holds for any $a \in \mathbb{A}$ and $k \in \mathbb{N}$. So the theorem gives us functions $\overline{f}_\sigma \in \mathsf{T}_\alpha(\Sigma)_\sigma \to_{\mathrm{fs}} \mathbb{N}$. Writing $\mathit{length}\, e$ for $\overline{f}_\sigma\, e$, we have the expected properties of a length function on α-terms:

$$\mathit{length}\, a = 1 \qquad \mathit{length}(e_1, \ldots, e_n) = \mathit{length}\, e_1 + \cdots + \mathit{length}\, e_n$$
$$\mathit{length}(K\ e) = \mathit{length}\, e + 1 \qquad \mathit{length}(a.\, e) = \mathit{length}\, e + 1\ .$$

Note that the last clause holds for all a, because in (18) the condition "$a \notin A$" is vacuously true (since $A = \emptyset$).

Example 15 (Recursion with "varying parameters"). Norrish [14, p 245] considers a variant $\mathtt{sub}'$ of capture-avoiding substitution whose definition involves recursion with varying parameters; it motivates the parametrised recursion principle he presents in that paper. The α-structural recursion principles we have given here do not involve parameters, let alone varying ones; nevertheless it is possible to derive parameterised versions from them. One can derive parameterised versions of ordinary structural recursion by currying parameters and defining maps into function sets using Theorem 2. In the presence of binders, one has to do something slightly more complicated, involving the Freshness Lemma 7, to derive the parameterised FCB from the unparameterised version of the condition.

Let us see how this works for Norrish's example. Using the nominal signature Σ^λ from Sect.4.3 (for which $\mathsf{T}_\alpha(\Sigma)_{\mathsf{t}}$ coincides with the nominal set $\Lambda/{=_\alpha}$ of α-equivalence classes of λ-terms) his sub' function can be expressed as follows. Fixing atoms $a_1, a_2 \in \mathbb{A}_{\mathsf{v}}$, we seek a function $s \in (\Lambda/{=_\alpha}) \to_{\mathrm{fs}} (\Lambda/{=_\alpha}) \to_{\mathrm{fs}} (\Lambda/{=_\alpha})$ satisfying:

$$s\,(\mathit{Var}\, a)\, e = \mathit{if}\ a = a_1\ \mathit{then}\ e\ \mathit{else}\ \mathit{Var}\, a \tag{22}$$
$$s\,(\mathit{App}(e_1, e_2))\, e = \mathit{App}(s\, e_1\, e, s\, e_2\, e) \tag{23}$$
$$a \# (a_1, a_2, e)\ \Rightarrow\ s\,(\mathit{Lam}\, a.\, e_1)\, e = \mathit{Lam}\, a.\, s\, e_1\, (\mathit{App}(\mathit{Var}\, a_2, e))\ . \tag{24}$$

If can be obtained from Corollary 12 as $s = \overline{h}$ if we take X to be the nominal set $(\Lambda/{=_\alpha}) \to_{\mathrm{fs}} (\Lambda/{=_\alpha})$ and use the functions

$$h_{\mathit{Var}} \triangleq \lambda a \in \mathbb{A}_{\mathsf{v}}.\lambda e \in (\Lambda/{=_\alpha}).\ \mathit{if}\ a = a_1\ \mathit{then}\ e\ \mathit{else}\ \mathit{Var}\, a$$
$$h_{\mathit{App}} \triangleq \lambda(x_1, x_2) \in X \times X.\ \lambda e \in (\Lambda/{=_\alpha}).\ \mathit{App}(x_1\, e, x_2\, e)$$
$$h_{\mathit{Lam}} \triangleq \lambda(a, x) \in \mathbb{A}_{\mathsf{v}} \times X.\lambda e \in (\Lambda/{=_\alpha}).\ \mathit{fresh}(h(a, x, e))$$

[10] The same goes for Norrish's `stripc` function, used to illustrate the limitations of Gordon and Melham's workaround for the *length* function [14, p. 247].

where the last clause uses Lemma 7 applied to $h(a,x,e) \triangleq \lambda a' \in \mathbb{A}_{\mathrm{v}}.\ \mathit{Lam}\, a'.\, ((a\ a') \cdot x)(\mathit{App}(\mathit{Var}\, a_2, e)) \in \mathbb{A}_{\mathrm{v}} \to_{\mathrm{fs}} (\Lambda/{=_\alpha})$, which is easily seen to satisfy the property (13) needed for to apply lemma. Properties (19) and (20) of $\overline{h}$ give (22) and (23) respectively. When $a \neq a_1, a_2$, property (21) gives us $\overline{h}(\mathit{Lam}\, a.\, e_1) = h_{\mathit{Lam}}(a, \overline{h}\, e_1) = \mathit{fresh}(h(a, \overline{h}\, e_1, e))$. So if $a \# (a_1, a_2, e)$, picking any $a' \# (a_1, a_2, e, e_1, h)$, then by Lemma 7 we have $\mathit{fresh}(h(a, \overline{h}\, e_1, e)) = h(a, \overline{h}\, e_1, e)\, a' \triangleq \mathit{Lam}\, a'.\, ((a\ a') \cdot (\overline{h}\, e_1))(\mathit{App}(\mathit{Var}\, a_2, e)) = \mathit{Lam}\, a'.\, (a\ a') \cdot (\overline{h}\, e_1\, (\mathit{App}(\mathit{Var}\, a_2, e)))$. Hence by definition of $=_\alpha$, $\overline{h}(\mathit{Lam}\, a.\, e_1) = \mathit{Lam}\, a.\, \overline{h}\, e_1\, (\mathit{App}(\mathit{Var}\, a_2, e))$, as required for (24).

6 Assessment

Mathematical Perspective. The results of this paper are directly inspired by my joint work with Gabbay on "FM-set" theory [8] and by his PhD thesis [6]; in particular those works contain structural recursion and induction principles for an inductively defined FM-set isomorphic to λ-terms modulo α-equivalence. Here I have taken an approach that is both a bit more general and more concrete: more general, because the particular signature for λ-terms has been replaced by an arbitrary nominal signature (a notion which comes from joint work with Urban and Gabbay [22] and is developed further in Cheney's thesis [2]); and more concrete in two respects. First, the key notion of (finite) *support* has been developed using nominal sets within the framework of ordinary higher-order logic, rather than being axiomatised within FM-set theory; see Cheney [2, Chapter 3] for a more leisurely and generalised account of the theory of nominal sets. Secondly, rather than using an inductively defined nominal set that is isomorphic to the set of α-terms, the recursion and induction principles refer directly to α-terms, i.e. standard α-equivalence classes of abstract syntax trees. This is also the approach taken by Norrish [14], building on Gordon and Melham's five axioms for α-equivalence [9]; and also by Urban and Tasson [23]. Norrish's recursion principle [14, Fig. 1] has side-conditions requiring that the function being defined be well-behaved with respect to variable-permutations and with respect to fresh name generation. In effect these side-conditions build in just enough of the theory of nominal sets to yield a well-defined and total function, while only having to specify how binders with fresh names are mapped by the function. Along with Urban and Tasson [23], I prefer to develop the theory of nominal sets in its own right and then give a simple-looking (compare the statements of Theorems 2 and 8) recursion principle within that theory. One advantage of such an approach is that it makes it easier to identify and use properties of name *freshness*, such as Lemma 7, independently of the recursion principle. We used Lemma 7 in the reduction of Theorem 8 to Theorem 2 and in the reduction of "varying parameters" to "no parameters" (Example 15); another good example of its use occurs (implicitly) in the denotational semantics of FreshML's `fresh` expression [20, Sect. 3].

Automated Theorem-Proving Perspective. How easy is it to apply these principles of α-structural recursion and induction? Just as for the work of Gordon-Melham, Norrish and Urban-Tasson, to use them one does not have to change to an unfamiliar logic (we remain in higher-order logic), or a new way of representing syntax (we use the familiar notion of α-equivalence classes of abstract syntax trees). One does have to get used

to thinking in terms of permutations and finite support; and the latter is undoubtedly a subtle concept at higher types. However, the relativisation from arbitrary mathematical objects to finitely supported ones called for by this approach is made easier by the fact, noted in Sect. 4.4, that the finite support property is conserved by all the usual constructs of higher-order logic except for uses of the axiom of choice. Based also on my experience with other formalisms, I claim that the use of permutations and finitely supported objects is a simple, effective and yet rigorous way of dealing with binders and α-equivalence in "paper-and-pencil" proofs in programming language semantics. But how easy is it to provide computer support for reasoning with α-structural recursion and induction? In Sect.4.4, I mentioned the three types (I–III) of task involved in applying these principles in any particular case. Task (III) will require human-intervention; but in view of the meta-theorem mentioned in Sect.4.4, there is the possibility of making tasks (I) and (II) fully automatic. One way of attempting that is to develop a new higher-order logic in which types only denote nominal sets and that axiomatises properties of permutations and finite support; this is the route taken by Gabbay with his FM-HOL [7]. The disadvantage of such a "new logic" approach is that one does not have easy access to the legacy of already-proved results in systems such as HOL (`hol.sourceforge.net`) and Isabelle/HOL (`www.cl.cam.ac.uk/Research/HVG/Isabelle/`). To what extent tasks (I) and (II) can be automated within these "legacy" mechanised logics remains to be seen. The work of Norrish [14] provides a starting point within the HOL system; and Urban and Tasson [23] have already developed a theory equivalent to nominal sets within Isabelle/HOL up to and including what I am here calling α-structural induction for the particular nominal signature for λ-terms (but not yet α-structural recursion for that signature).[11] HOL and Isabelle/HOL feature type variables and predicative polymorphism. As a result, in principle it is possible to formulate and prove within such logics a result like Theorem 8 that makes a statement about *all* nominal signatures and *all* nominal sets. Of more use in practice would be would an augmentation of the HOL or Isabelle datatype packages, allowing the user to declare a nominal signature and then have the principles of α-structural recursion and induction for that signature proved and ready to be applied.[12]

Acknowledgements. I am grateful to James Cheney, Murdoch Gabbay, Michael Norrish, Mark Shinwell and Christian Urban for their many contributions to the subject of this paper.

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
[2] J. Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, August 2004.
[3] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[11] Their proof of validity of the induction principle follows a different route from the one used here to prove Theorem 8 and Corollary 10.

[12] How best in such a package to deal with the relativisation to nominal sets is certainly an issue; Urban and Tasson report that Isabelle's *axiomatic type classes* are helpful in this respect.

[4] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.
[5] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. LICS'99*, pages 193–202. IEEE Computer Society Press, 1999.
[6] M. J. Gabbay. *A Theory of Inductive Definitions with α-Equivalence: Semantics, Implementation, Programming Language*. PhD thesis, University of Cambridge, 2000.
[7] M. J. Gabbay. FM-HOL, a higher-order theory of names. In F. Kamareddine, editor, *Workshop on Thirty Five years of Automath, Informal Proceedings*. Heriot-Watt University, Edinburgh, Scotland, April 2002.
[8] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
[9] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In *Proc. TPHOLS'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–191. Springer-Verlag, 1996.
[10] T. G. Griffin. Notational definition — a formal account. In *Proc. LICS'88*, pages 372–383. IEEE Computer Society Press, 1988.
[11] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
[12] F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *Proc. ICALP 2001*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer-Verlag, 2001.
[13] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.
[14] M. Norrish. Recursive function definition for types with binders. In *Proc. TPHOLS 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 241–256. Springer-Verlag, 2004.
[15] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. PLDI'88*, pages 199–208. ACM Press, 1988.
[16] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
[17] G. D. Plotkin. An illative theory of relations. In R. Cooper, Mukai, and J. Perry, editors, *Situation Theory and its Applications, Volume 1*, volume 22 of *CSLI Lecture Notes*, pages 133–146. Stanford University, 1990.
[18] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.
[19] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
[20] M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 2005. To appear.
[21] A. Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.
[22] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.
[23] C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In *Proc. CADE-20*, *Lecture Notes in Computer Science*, Springer-Verlag, 2005.

Shallow Lazy Proofs

Hasan Amjad

University of Cambridge Computer Laboratory, William Gates Building,
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
Hasan.Amjad@cl.cam.ac.uk

Abstract. We show that delaying fully-expansive proof reconstruction for non-interactive decision procedures can result in a more efficient workflow. In contrast with earlier work, our approach to postponed proof does not require making deep changes to the theorem prover.

1 Introduction

Theorem proving programs serve to mechanise formal reasoning. Interactive theorem provers require user guidance to assist with formal proof. The proof proceeds by having the user type commands that tell the prover the next step in the proof. Such a step may invoke fully automatic decision procedures to handle the tedious but easier parts of the proof, leaving the user to focus on the parts requiring insight. Such decision procedures can often take a while to execute, taking up resources and making the user wait before the next command can be issued.

This situation is made worse for *fully-expansive* or *LCF-style* [10] theorem provers. This is because every proof must be constructed via the application of a small core set of primitive rules of inference. This has the great advantage that writing more powerful derived rules of inference cannot breach soundness, provided the core rules are sound. However, there is a speed penalty because of the large number of primitive inferences performed for each proof. Henceforth we consider only fully-expansive provers.

The standard way of making a decision procedure go faster is to execute it outside the theorem prover in an efficient manner (say as a C program), and then verify the result by proof in the theorem prover. Often the difference between the time taken to find a proof and to verify it justifies this approach. For instance, the result from a SAT-solver can be efficiently verified fully-expansively [7]. On the other hand, the result of a BDD operation is extremely inefficient to verify fully-expansively [11]. Most decision procedures fall somewhere along this spectrum.

For decision procedures that are hard to verify, the proof is usually done using a hybrid approach in which as much as possible of the proof search is done externally, resorting to internal proof if the trade-off between search and verification becomes unacceptable. In this way, proof search and verification proceed together in lock-step fashion.

In such a situation, it may help to postpone the justification underlying the verification, but to assert the verification right away so that the next search phase can proceed. This is supported by two observations:

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 35–49, 2005.

- The verification can be done later when the user is otherwise occupied and computational resources are lying unused.
- Many of the lemmas for the verification proof may not eventually be required as the decision procedure may abandon a particular line of inquiry and thus all corresponding verification as well.

This approach was considered, most relevantly for us, by Boulton in his Ph.D. thesis [4], for the HOL88 theorem prover. The postponed proofs and theorems were termed *lazy*. In this paper we describe a slightly different approach to lazy proofs, which is less general but nonetheless useful in certain circumstances. We use the HOL-4 system [13,9], a descendant of HOL88.

2 Background

To understand our approach, it is important to first know about the inference system of the HOL family of provers.

2.1 The HOL Inference System

We first give a quick overview of the inference system of HOL88 and HOL-4.[1] The logic of the prover is classical higher order logic, i.e. Church's simple type theory [5], with Hindley-Milner polymorphism [16]. The inference system is based on natural deduction, though the choice of exactly which rules are primitive has been tempered by efficiency considerations. Hence some derivable rules are implemented as primitives. The term structure is simply-typed λ-calculus and the formula syntax is that of predicate logic.

The system is implemented in Moscow ML [19], a dialect of Standard ML [17]. The type of terms is `term` and the ML type constructors are available to users. A *goal* is an unproved theorem, consisting of a set of propositional terms Γ (the assumptions) and a propositional term A (the conclusion). Thus a goal has ML type $\texttt{term}\,\texttt{set} \times \texttt{term}$. A theorem $\Gamma \vdash A$ is a goal that can be derived via the primitive inference rules. Theorems have ML type `thm`. This is an abstract type, and so theorems can only be constructed by using the inference rules which are implemented as ML functions. Some of the primitive rules of inference we shall refer to are shown in Figure 1.

The complete set of primitive rules together with other functions that may affect soundness is often referred to as the *kernel* of the prover. The kernel is trusted code in the sense that a bug in the kernel can affect the soundness of derivations. Hence fully-expansive provers attempt to keep the kernel as small as is feasible.

[1] The differences are irrelevant to our discussion so we may regard the inference system as effectively being the same. For instance, HOL88 was actually implemented in Classic ML, but for the sake of clarity we'll pretend it was written in Moscow ML.

Assumption:

$$\frac{}{\{A\} \vdash A}\mathsf{ASSUME}$$

Modus ponens (A and A' are α-convertible):

$$\frac{\Gamma \vdash A \quad \Delta \vdash A' \Rightarrow B}{\Gamma \cup \Delta \vdash B}\mathsf{MP}$$

Discharge (Δ is the (possibly empty) set of all terms in Γ α-convertible to B):

$$\frac{\Gamma \vdash A}{\Gamma - \Delta \vdash B \Rightarrow A}\mathsf{DISCH}B$$

Generalisation (x is not free in the assumptions Γ):

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\mathsf{GEN}\,x$$

Specialisation (y is free for x in A):

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[y/x]}\mathsf{SPEC}y$$

Fig. 1. Some of the primitive inference rules of HOL-4

2.2 Deep Lazy Proofs

Since Boulton's method forms the starting point of our investigations, we provide an overview of it here. We shall call this method *deep* lazy proofs to distinguish it from our approach. The idea behind deep lazy proofs is to replace theorems with lazy theorems of abstract type `lazy_thm`.

The lazy theorems are lazy because they have not yet been proved, but may be used in proofs. Each consists of the underlying structure (i.e. the goal) of the theorem, together with a *justification function* of ML type `unit → thm`. The justification function or *justifier*, is a closure which when executed returns a proved theorem of type `thm`. A lazy theorem is created by calling a special function `mk_lazy_thm` that takes a goal and a justifier and returns a lazy theorem.

This requires modifying all inference rules (including those in the kernel) to operate over `lazy_thm` rather than `thm`. The lazy inference rules would perform the required validity checks, internally modify the goal, and construct a new justifier that sequentially calls the justifiers of the argument lazy theorems before creating the required theorem.

This way any lazy theorem can be proved by executing its justifier. This execution will also result (because of the sequencing) in the execution of any justifiers for lazy theorems that were arguments to the inference rules that gave rise to the lazy theorem. A failed justification raises an exception. This ensures that all lazy theorems used in the lazy proof of the lazy theorem under consideration are indeed provable. The function `prove_lazy_thm` takes a lazy theorem, executes the justifier, checks that the theorem it returned is indeed the goal, and returns it as a proved theorem of type `thm`.

Since the entire kernel is now lazy, all proof can be done using lazy theorems only. This results in a significant speed up since simply modifying the term structure of the goal and sequencing closures is in general much faster than calling an eager inference rule. This does not really apply at the level of primitive lazy inferences which effectively do as much work as their eager counterparts. The main benefit is for derived rules that can directly create lazy rules whose justifiers take a while to execute.

Proved theorems of type `thm` are still present in the system mainly because HOL88 was unable to persist closures, so lazy theorems had to be proved before theories could be saved between sessions. To allow these theorems to be reused in the main system, the type of lazy theorems was modified to optionally contain a proved theorem. A function was provided to convert a value of type `thm` to a value of type `lazy_thm`, the latter simply containing the proved theorem. This would enable saved theorems to be read back into the system in later sessions.

The deep approach has the obvious advantage of increased apparent speed (i.e. not including the justification time). Correctly used, it also results in increased real speed as lazy inferences applied in unused branches of proof search are not needlessly justified. Another big advantage is that the chosen representation does not impose any restrictions: deep lazy theorems can be used anywhere in the prover (except for storage).

The disadvantages are primarily from a developer's perspective. First, this approach requires modifying the kernel, always a delicate enterprise in a fully-expansive prover. Second, unmodified derived rules do not become faster automatically. They have to be heavily modified to exploit lazy proof, and there is no guarantee that total execution time (i.e. lazy part plus justification) will be lower than the standard eager proof: it depends on the nature of the proof search the decision procedure performs.

Henceforth, we use the word "theorem" to mean both lazy and proved theorems, providing the appropriate qualification when necessary. We use the words "lazify" and "unlazify" to denote the operations of constructing a lazy theorem, and recovering a proved theorem from it.

3 A Shallow Approach

We felt the need for postponed proof during the development of a symbolic model checker embedded in HOL-4 [1]. The checker falls firmly in the "hard to verify" category, with BDD operations interleaved with proof steps. Since the model checker is completely non-interactive, being able to postpone expensive verification to, say, when the user is asleep,[2] or do it in the background, would help significantly with the work-flow and some of the verification could possibly be discarded unused.

[2] Conversations with verification engineers at Intel Corp. and Microsoft Corp. indicate that model checkers are commonly run overnight, with office time spent setting up runs or analysing the output.

However, it is not clear whether deep lazy theorems are the answer. In particular, we do not wish to modify the kernel since we wish to retain compatibility with HOL-4. Hence the word "shallow" to contrast with Boulton's method. This single restriction starts a chain of decisions that results in a rather different approach to lazy proofs.

For a start, we cannot have a separate type to represent lazy theorems. This is because the primitive rules operate over normal theorems, and thus so do all the derived rules. Having lazy derived rules that operate over a different type would be an interoperability nightmare. Whereas these derived lazy rules could conceivably use normal theorems via a suitable embedding of normal theorems into lazy ones (much like in the deep approach), the reverse is not possible unless the lazy theorem is first justified into a normal one. This would defeat the point of having lazy theorems, or else severely restrict their usefulness.

Also, since justifiers can now not be stored with the theorems (since we cannot modify the type `thm`), they have to be stored centrally, and a mapping ϕ between theorems and justifiers maintained. ϕ is a partial function, since a single theorem is not allowed two different justifiers, and proved (directly or unlazified) theorems have no justifiers. In ML, we implement ϕ by the look-up function on a splay map data structure.

3.1 Creating Lazy Theorems

Our solution[3] is to have a lazy theorem for the goal (Γ, A) be represented by the theorem $\{A^g\} \vdash A$, where $A^g \equiv \forall x \in fv(A).\, A$ and fv computes the set of free variables of A. The mapping to justifiers is then given via terms by constructing ϕ so that $\phi(A^g)() = \Gamma \vdash A$. The only constraint is that

$$fv(A) \cap \bigcup_{\gamma \in \Gamma} fv(\gamma) = \emptyset$$

i.e., the conclusion and the assumptions must not have any free variables in common. The reason for this is discussed towards the end of §3.2.

$\{A^g\} \vdash A$ is derived by first constructing the term A^g and then proceeding

$$\begin{array}{l} \dfrac{}{\{A^g\} \vdash A^g}\ \mathsf{ASSUME} \\ \quad \vdots\ \mathsf{SPEC}(x \in fv(A)) \\ \{A^g\} \vdash A \end{array} \tag{1}$$

where after the first inference we have a series of applications of the SPEC rule for each $x \in fv(A)$.

Users are not allowed to modify ϕ directly. A function `make_lthm` is supplied that takes a goal, creates the required mapping in ϕ, and returns the lazy theorem. §3.3 demonstrates this on a simple example.

[3] We note here that our first attempt based on the `mk_oracle_thm` function of HOL-4 failed because it was unable to cleanly track primitive inference rule applications.

The reason we do not retain the assumptions Γ is practical: it does no harm, and retaining them complicates and slows down look-ups when evaluating ϕ. As noted in the next section, these assumptions are recovered from the justifier when the theorem is unlazified.

The reason for `make_lthm` creating $\{A^g\} \vdash A$ rather than $\{A\} \vdash A$ is to ensure that rules like GEN that would have succeeded on the proved theorem $\Gamma \vdash A$ do not fail on the lazy version. Indeed, some rule applications that would have failed on $\Gamma \vdash A$ succeed on $\{A^g\} \vdash A$. This would appear to suggest that we can prove stronger theorems with $\{A^g\} \vdash A$ than we could with $\Gamma \vdash A$. However, all such rule applications turn out to be vacuous (e.g. $\mathsf{GEN}\, x\, t$ where x does not occur in t), so the "stronger" theorem is in fact derivable from $\Gamma \vdash A$.

3.2 Unlazifying Lazy Theorems

Now suppose we wish to prove B, and know that we require only $\Gamma \vdash A$ to prove it. We can use $\{A^g\} \vdash A$ instead of $\Gamma \vdash A$, since the former is a stronger theorem. The resulting "lazy" theorem is then $\{A^g\} \vdash B$.

In general, to unlazify a theorem $\Delta \vdash B$, we define the operation $(-)^s$ on terms to be such that $(A^g)^s = A$ and for each $C \in \Delta$ such that $\phi(C)$ is defined, do

$$\dfrac{\dfrac{\Delta \vdash B}{\Delta - C \vdash C \Rightarrow B}\ \mathsf{DISCH}\, C \qquad \begin{array}{l}\phi(C)() \\ = \Theta \vdash C^s \\ \vdots\ \mathsf{GEN}\,(x \in fv(C^s)) \\ \Theta \vdash C\end{array}}{(\Delta - C) \cup \Theta \vdash B}\ \mathsf{MP} \tag{2}$$

If ϕ is not defined, we move on to the next $C \in \Delta$ and repeat. We can access the contents of Δ since type destructors for `thm` are available.

Thus we eventually derive $\Delta' \cup \Theta' \vdash B$ as required, where Δ' contains all the *non-lazy* assumptions used in the proof of B, i.e. they were not introduced via `make_lthm`. Θ' similarly contains all non-lazy assumptions used in the proofs of the lazy theorems used in the proof of B.

As an example, consider the lazy theorem $\{A^g\} \vdash B$ described at the beginning of §3.2. Δ in this case is just $\{A^g\}$. Iterating over Δ, we consider $\{A^g\}$, and compute $\phi(A^g)()$ which gives $\Gamma \vdash A$, corresponding to $\Theta \vdash C^s$ in derivation 2 above. Now we generalise this by apply GEN for each $x \in fv(A)$, to get $\Gamma \vdash A^g$, corresponding to $\Theta \vdash C$ above. Now we take our lazy theorem $\{A^g\} \vdash B$ and use DISCH to get $\vdash \{A^g\} \Rightarrow B$, corresponding to $\Delta - C \vdash C \Rightarrow B$ above. Finally we use MP together with $\Gamma \vdash A^g$ to obtain $\Gamma \vdash B$, corresponding to $(\Delta - C) \cup \Theta \vdash B$. We give a more detailed and concrete example in §3.3.

A function `prove_lthm` is supplied that takes a theorem and discharges by the above strategy all assumptions in Δ that were added due to invocations of `make_lthm`. The implementation of `prove_lthm` is a recursion rather than a simple iteration, since C^s may itself be the conclusion of a lazy theorem.

`prove_lthm` also modifies ϕ so that the next call $\phi(C)()$ will just return the result of the justification. This ensures that each unique lazy theorem's justifi-

cation is carried out at most once. This is critical since often some fundamental lemmas are used hundreds of times in a single run of the decision procedure.

Note that the series of GEN rule applications will fail if any $x \in fv(C^s)$ occur free in Θ. This is the reason for having the constraint on free variables in §3.1. The assumptions added by `make_lthm` respect this constraint so lazy theorems may be used by the justifiers of other lazy theorems.

Note also that dropping the assumptions Γ in the goal of a lazy theorem during `make_lthm` does not matter, as any of them that do not get discharged (as they would if they were caused by a yet earlier `make_lthm` invocation), will appear as part of Θ' in the final theorem.

3.3 A Simple Example

We give a simple concrete example to illustrate our approach. Suppose we wish to prove that

$$\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y + z = x + z)$$

This can be proved directly using the decision procedures for linear arithmetic available in HOL-4. For the purposes of this example however, another way to prove this in HOL-4 is to use the simplifier together with the theorem

$$\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y = x)$$

Suppose the latter theorem can be proved by a HOL-4 proof procedure f that takes the goal as argument and returns the required theorem. Suppose further that calling f is expensive.

To postpone the expensive calls, we can create a lazy theorem. This is done by calling `make_lthm` with the goal $goal_1$ given by

$$(\{'\forall m.m + 0 = m'\},'(y = 0) \Rightarrow (x + y = x)')$$

and the justifier $\lambda\_.\, f(goal_1)$.[4] `make_lthm` returns the lazy theorem

$$\{\forall xy.(y = 0) \Rightarrow (x + y = x)\} \vdash (y = 0) \Rightarrow (x + y = x)$$

and modifies ϕ so that $\phi('\forall xy.(y = 0) \Rightarrow (x + y = x)') = \lambda\_.\, f(goal_1)$. Note that the assumption set $\{'\forall m.m + 0 = m'\}$ has been dropped.

Now we prove our required theorem via simplifying with the lazy theorem, and get

$$\{\forall xy.(y = 0) \Rightarrow (x + y = x)\} \vdash (y = 0) \Rightarrow (x + y + z = x + z)$$

Later, to unlazify this theorem, we pass it to `prove_lthm`. `prove_lthm` iterates through the assumptions. In this case there is only one: $'\forall xy.(y = 0) \Rightarrow (x + y = x)'$.

[4] We use quotes to distinguish terms of HOL-4's object logic from our metatheory. There is no such danger of confusion for theorems, since all theorems in this section are in the object logic.

This is passed to ϕ which duly returns $\lambda\_.\, f(goal_1)$. Executing this closure gives us $\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y = x)$. `prove_lthm` now recursively calls itself with this theorem, in case it is lazy. In this example it is not, so the recursive call returns the theorem itself. We now repeatedly use GEN to get

$$\{\forall m.m + 0 = m\} \vdash \forall xy.(y = 0) \Rightarrow (x + y = x)$$

Note that the dropped assumption has reappeared via the justifier.

Finally we use DISCH $'\forall xy.(y = 0) \Rightarrow (x + y = x)'$ on our target theorem to get

$$\vdash (\forall xy.(y = 0) \Rightarrow (x + y = x)) \Rightarrow (y = 0) \Rightarrow (x + y + z = x + z)$$

and then use the MP rule on this together with $\{\forall m.m+0=m\} \vdash \forall xy.(y=0) \Rightarrow (x + y = x)$ from the previous paragraph to obtain

$$\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y + z = x + z)$$

as required.

3.4 Limitations and Benefits

This approach has a few problems not faced by the deep approach, which we discuss here:

1. Definitions cannot be lazified. This is because creating a definition introduces a fresh constant into the current theory, invalidating similarly named constants declared earlier. Thus the justifier's result will not match the corresponding assumption in the theorem to be unlazified and failure will occur at the MP stage of derivation 2.

   This is not such an issue because few decision procedures create definitions on-the-fly. If they do, the definitions are typically not recursive (used for abbreviating bigger terms say) and can be created very quickly.
2. The constraint in §3.1 means that our method cannot, in general, be used for interactive proof, except within a decision procedure that returns non-lazy results. However, often laziness in decision procedures allows some savings compared to eager execution, so its usefulness for interactive proofs is not entirely ruled out. Outside of interactive proof, this constraint has not been a problem in our experiments so far i.e. we have never needed to create a lazy theorem with assumptions that had free variables in common with the conclusion.
3. Any type variables in the conclusion of the lazy theorem appear in its assumptions as well. In theory, this is not a problem, since such type variables can be instantiated at the point of use. In practice, many derived rules in HOL attempt to instantiate type variables in the conclusion only and will fail with a lazy theorem which has those type variables in the assumptions (they must fail: unsoundness would result otherwise).

   The only complete answer to this is to modify all derived rules to attempt full type instantiation whenever possible. This could result in loss of

efficiency. Fortunately, most decision procedures tend to work mostly with ground theorems. Any theorems with type variables are usually general ones from pre-proved libraries. Such theorems are of course fully justified theorems and the problem does not exist.

4. ϕ is currently not constrained to be injective. This can create undesirable additional assumptions in an unlazified theorem, because the wrong justifier got called. Unlazifying would succeed since the justifier's result's conclusion is the desired one, but the final unlazified theorems would be stuck with the non-lazy assumptions of the justifier's result.

   This is an engineering issue. Making ϕ injective will require the mapping translation to take assumptions into account, which could possibly be slow. In practice, this turns out to be a low priority issue since in all our test runs so far we have yet to come across a non-injective ϕ.
5. The series of GEN applications in derivation 2 can be very expensive if Θ contains very large terms.
   This is a difficult problem. In practice, decision procedures often work with large terms, since the very reason for fully automated tools is to handle mountains of trivial proof. This problem can be solved by adding an iterated version of GEN to the kernel and deriving GEN as a non-primitive inference rule. We have verified this solution experimentally. However, modifying the kernel is not to be taken lightly. Fortunately, this does not become a big issue for our particular application (the embedded model checker) so we are content to live with it.

With the possible exception of the last, these are not really problems in practice, at least not for the problem domains we have considered so far. Unexpectedly, the fact that theorems tend to carry around a lot of assumptions while in lazy mode does not seem to have impacted performance. This may be because for our current problem domain, the number of dischargeable assumptions in the final lazy theorem rarely exceeds a hundred.

Within these constraints, we are able to create and use lazy theorems more or less as in the deep approach. We do have a few benefits as well:

1. The kernel is unmodified and so we are guaranteed soundness (modulo soundness of the kernel itself). This was a big plus during development when the ramifications of the method were still unclear.
2. No existing derived rule (or any other HOL-4 code for that matter) needs to be modified at all to work with this method. However, as with the deep approach, unmodified derived rules do not directly benefit.
3. There is only one type of theorem. Thus we do not have to implement translations between different types.

   More importantly, we can unlazify a theorem at any stage. This provides some flexibility in implementing a more sophisticated system that would perhaps completely or partially unlazify theorems based on some metric that measured the trade-offs for retaining the lazy version, at any stage during the decision procedure's run. The closest analogy that comes to mind is that of a garbage collector in modern programming languages.

3.5 Refinements

We mention some refinements to the method which, for the sake of clarity, were not presented in §3.1 and §3.2.

First, as with the deep approach, we allow different modes of operation. Lazy mode works as described above. In eager mode, the justifiers are executed right away so there is no difference between eager mode and normal non-lazy execution. The mode can be changed at any time, and thus procedures can switch lazy proofs on or off depending on requirements.

Second, `make_lthm` is actually passed two closures rather than a goal and a single closure (i.e. the justifier). The first closure returns the goal, and the second is the justifier. This is so that eager mode does not pay the penalty for constructing the goal (which can sometimes be time consuming even in lazy mode), which is not required in eager mode. This does not guard against the goal being different from the justifier's result. However, if such were the case, unlazifying would fail at the MP stage of derivation 2, if not earlier.

Finally, the first closure passed to `make_lthm` does not just return a goal. It returns a goal and optionally a justifier which is used when unlazifying. This is because it can use any information gained during the construction of the goal in lazy mode. Thus it could conceivably execute faster than the "eager" justifier (the second argument to `make_lthm`) that re-proves everything from scratch.

We have experimented with using shallow lazy proofs in our embedded model checker. The results are promising, as we show in the next two sections.

4 Lazifying a Decision Procedure

We chose to modify the HolCheck decision procedure [1] in HOL-4 to exploit lazy proof. The reason for this is our view that HolCheck could significantly benefit from such a development, as well as our familiarity with the code.

HolCheck is a symbolic model checker for the modal μ-calculus [15]. It is embedded in the HOL logic, with BDD operations considered atomic and executed externally for efficiency [8]. It is not possible to efficiently verify the part of proof search performed by the BDD operations. However, a model checker does more than BDD operations. It creates and applies transformations to models, translates between formalisms, and organizes the sequencing of BDD operations. All these operations are amenable to verification by formal proof.

Since model checkers are expected to be fast, it is in our interest to lower the overhead that is the verification part of the execution of HolCheck. For various reasons, it was found more efficient and more effective to carry out the verification interleaved with the proof search steps. The expectation is that by lazifying the verification, we retain the advantages and yet increase efficiency.

The model checking work-flow is somewhat different from the way interactive proof proceeds. The user sets up a run, and executes the checker. Most of the user's time is then spent analysing the output of the checker to fix bugs in the model or the specification. Then the process is repeated. It is thus imperative that the checker run as fast as possible.

Our main hope was that by using lazy proofs, we could delay the relatively slower verification parts of the proof to some later time, say when the user is analysing the checker's output, or overnight. This would greatly speed up the model checking work-flow, at no extra cost.

Currently we have lazified only the core of the model checker. This includes formal model construction, translation by proof from the specification logic CTL [3] to the model checker's μ-calculus and back, the model checking engine itself and trace generation. The abstraction mechanisms have not yet been made lazy, and were turned off for benchmarking purposes.

The benchmarks are for model checking several properties for a 3x3 game of tic-tac-toe, and for a small pipelined ALU. The examples were chosen to be big enough to not finish execution in a couple of seconds, but not so big that the majority of the model checker's time was spent in BDD operations. This was so that we could time the savings (if any) in the non-BDD verification part of the checker's time.

Table 1. HolCheck benchmarks with and without lazy proof (as % of eager time)

	tic-tac-toe	ALU
Eager total (gc)	100 (30.13)	100 (34.28)
Lazy lazy	38.88	30.47
Lazy justification (overhead)	19.53 (7.39)	75.95 (4.98)
Lazy total (gc)	58.41 (6.21)	106.43 (4.76)

The results are given in Table 1. All numbers are normalised with respect to the topmost row, which represents the time taken by running HolCheck in eager or non-lazy mode, scaled to 100. The absolute numbers are not important. The benchmarks took from about 30 seconds to about a 1000 seconds to complete.

In the table, "gc" stands for the time taken by the Moscow ML garbage collector, and is included in the total time. "lazy" gives the time taken to run HolCheck in lazy mode, and "justification" indicates how long it takes to later unlazify all the lazy theorems produced. "overhead" shows the portion of time spent doing the work of derivations 1 and 2.

The HolCheck core's verification part is already heavily optimized. Nonetheless, we see that the lazified system runs around thrice as fast as the eager one. Garbage collection overheads are also comparatively low, mainly because proof is delayed so that the number of terms that appear and disappear is low.

Factoring in the justification stage, the tic-tac-toe benchmark still runs almost twice as fast as the eager version, but the ALU benchmark is slightly slower. This is because in the tic-tac-toe benchmark, some of the properties checked are designed to fail (to exercise the trace generation code). Hence unlazifying for those properties is not done, whereas the eager version expends extra time proving the corresponding verification conditions. In the ALU benchmark, all properties succeed, so all theorems must be unlazified. In this case, the small lag is explained by the (un)lazifying overhead, as expected.

Note that we omit the number of primitive inferences made in the different modes. This is mainly because the number of primitive inferences only very roughly correlates with time savings and thus primitive inference figures may overrate the benefits of lazy proofs. For instance, the lazy version of HolCheck performs about 80% fewer primitive inferences than the eager one (though the justification phase makes up for the difference), but is not five times as fast. Unlike the deep approach, our lazy mode is not primitive inference free: we do not bother lazifying proofs written directly in terms of primitive inferences since these do not save us any time. There were quite a few of these in the pre-lazified optimized HolCheck core.

The speed gains are remarkably similar to those achieved using the deep approach, which also reported roughly a three-fold increase in the speed of the lazy decision procedure, with justification not always saving any time. It remains to be investigated if this is more than coincidence. It should be kept in mind that the deep approach's results were reported over a decade ago, and theorem proving technology has improved much since then. Nonetheless, we have achieved similar performance gains without modifying the kernel of the prover, with room for improvement if we do.

5 Lazifying a Derived Rule

The primary beneficiaries of our approach are expected to be decision procedures. However, most decision procedures make use of derived rules of inference. It would thus be nice if we could over time build a library of lazified derived rules. This would ease the further development of lazy derived rules and decision procedures.

The expectation here is that the lazy derived rule will run faster than the eager one and even if the justification takes as long as the eager rule, it would not matter since the justification will be carried out at a time when the user is busy elsewhere.

An an experiment, we chose to lazify the GEN_PALPHA derived rule in HOL-4. This rule performs alpha conversion with respect to the binder of any abstraction, with the further ability to handle not only single variables, but paired structures as well. So,

$$\frac{\Gamma \vdash @x.t}{\Gamma \vdash @y.t[y/x]}\ \mathsf{GEN\_PALPHA}y$$

where @ is any binder, and x and y are possibly paired structures of the same type. Variable renaming is performed if necessary to avoid variable capture within t.

This rule was chosen for two reasons. First, it is used often in HolCheck and can be slow on large terms (not through any fault of the implementation). Second, it is not so low-level that lazifying has no benefit, and not so complicated that lazifying would not work because we do not understand how the rule works.

Lazifying this rule is not as simple as passing a goal and the eager version of GEN_PALPHA to make_lthm. We need to work out what the $t[y/x]$ part of

Table 2. GEN_PALPHA lazy execution time (as % of eager time)

Number of variables\Term size	800-3000	1500-5500	2500-8000	3000-12000
100	55.81	55.1	57.14	53.33
200	22.46	22.52	24.62	23.72
300	13.1	13.45	14.96	17.03
400	9.66	9.77	12.28	12.49
500	5.82	7.15	9.59	9.88

Table 3. GEN_PALPHA justification time (as % of eager time)

Number of variables\Term size	800-3000	1500-5500	2500-8000	3000-12000
100	188.37	289.8	422.98	572.08
200	150.97	203.95	266.36	314.71
300	135.1	172.79	202.24	227.16
400	125.69	152.15	171.57	190.51
500	128.49	144.26	154.56	165.87

the goal will look like. Simple substitution will not do since x and y may have the same type but different structure e.g. x could be the pair (a, b) with a and b occurring separately in t, while y is just a single variable. Also, some components of y may need renaming to avoid capture by internal binders in t. Fortunately, the code for this can be extracted without too much hardship from the code for GEN_PALPHA, though it is not as simple as a copy-paste operation.

Constructing the goal takes time, and this is where the motivation for the refinements mentioned in §3.5 comes from. Once that is done, we proceed with make_lthm as usual, using as our justifier the eager version of GEN_PALPHA. Later we intend to write our own justifier that can take advantage of any information gleaned during goal construction.

Performance evaluation results are given for lazy execution and justification in Table 2 and Table 3, with all times normalised to the eager version's performance, which is set at 100 as before.

Both the eager and lazy versions of GEN_PALPHA were run on a number of randomly generated terms of various sizes, with varying numbers of free variables and levels of internal binding (to exercise the renaming aspects of the code). Term size is, roughly speaking, the number of abstractions and applications in the underlying λ-structure of a term. Term sizes are given as ranges since the randomly generated terms contained predicates on a random number of variables, hence the term size increased with the number of free variables.

Again, the absolute numbers are not important. Just to give the reader an idea, the eager version averaged about half a second with a 100 variables and a term size range of 800-3000, and about 120 seconds when the number of variables is increased to 500. Increasing term size only causes a linear degradation in eager performance however.

As expected, the lazy version runs much faster than the eager version, much more so than in the deep approach. However, the justification phase now takes

much longer. The performance of the eager version scales well with term size, but degrades badly with increasing the number of variables, which is where the lazy version wins out. Justification degrades badly in both dimensions, though with higher variable sizes, the relative performance improves on account of the degradation in eager performance.

Since the justifier is just the eager version, all the extra time in justification is being taken up by the lazifying overhead. Profiling shows that 99% of this overhead can be traced to the series of GEN applications in derivation 2. This problem has already been noted. If an iterated version of GEN is added to the kernel, this overhead almost disappears.

Thus, at least in this case, using the lazy version is only useful if the justification is to be postponed until much later, rather than done immediately after the lazy rule finishes. This renders this lazy rule unfit for interactive use (without modifying the kernel) where, as noted in §3.4, only fully justified theorems are usable. However, it can still be used in decision procedures and indeed has been used in HolCheck.

6 Related Work and Conclusion

The work by Boulton [4] was the starting point of our research and his observations strongly influenced our decisions. Laziness has been exploited in formal verification elsewhere, for instance in incremental SAT-based abstraction for model checking [2,6]. More relevantly, the lazy approach has been applied to the Nelson-Oppen congruence closure algorithm [12,18].

Shallow lazy proofs succeed very well at justifying the observations made towards the end of §1, at least with respect to HolCheck and GEN_PALPHA. However, our experience with lazifying indicates that the gains are strongly dependent on the nature of the proof procedure under consideration.

In particular, the more proof is done externally, the less lazy proofs are likely to help, even in the lazy stage. Also, if the proof search is very efficient with few abandoned lines of search, the justification phase will make up for any time savings made during the lazy stage.

The constraints listed in §3.4 do restrict the applicability of our approach. However, in actual practice, the performance issues have not seriously affected efficiency, and the theoretical restrictions have not constrained shallow lazy proofs enough to affect usefulness.

This method is applicable to all theorem provers that support the rules of inference listed in Figure 1.

References

1. H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In David A. Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2003.

2. Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference*, volume 3114 of *LNCS*, pages 457–461. Springer, 2004.
3. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
4. Richard John Boulton. Efficiency in a fully-expansive theorem prover. Technical report, University of Cambridge Computer Laboratory, 1994.
5. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
6. Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer-Verlag, July 2002.
7. M. J. C. Gordon. `HolSatLib` documentation. `http://www.cl.cam.ac.uk/users/mjcg/HolSatLib/HolSatLib.ps`, 2002.
8. M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.
9. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : A theorem-proving environment for higher order logic*. Cambridge University Press, 1993.
10. M. J. C. Gordon, A. R. G. Milner, and C. P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. volume 78 of *LNCS*. Springer-Verlag, 1979.
11. J. Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38(2):162–170, 1995.
12. J. Harrison, 2005. Private communication. See directory `hol_light/Examples/holby.ml` of the HOL Light distribution [14].
13. The HOL-4 Proof Tool. Tool URL `http://hol.sf.net`, 2003.
14. The HOL Light Proof Tool. Tool URL `http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html`, 2005.
15. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
16. A. R. G. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
17. A. R. G. Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, May 1997.
18. George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In David A. McAllester, editor, *Conference on Automated Deduction (CADE)*, volume 1831 of *Lecture Notes in Computer Science*, pages 25–44. Springer, 2000.
19. Peter Sestoft. Moscow ML. `http://www.dina.dk/~sestoft/mosml.html`, 2003.

Mechanized Metatheory for the Masses: The PoplMark Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

Abstract. How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?

We propose an initial set of benchmarks for measuring progress in this area. Based on the metatheory of System $F_{<:}$, a typed lambda-calculus with second-order polymorphism, subtyping, and records, these benchmarks embody many aspects of programming languages that are challenging to formalize: variable binding at both the term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. We hope that these benchmarks will help clarify the current state of the art, provide a basis for comparing competing technologies, and motivate further research.

1 Introduction

Many proofs about programming languages are long, straightforward, and tedious, with just a few interesting cases. Their complexity arises from the management of many details rather than from deep conceptual difficulties; yet small mistakes or overlooked cases can invalidate large amounts of work. These effects are amplified as languages scale: it becomes hard to keep definitions and proofs consistent, to reuse work, and to ensure tight relationships between theory and implementations. Automated proof assistants offer the hope of significantly easing these problems. However, despite much encouraging progress in recent years and the availability of several mature tools (ACL2, Coq, HOL, HOL Light, Isabelle, Lego, NuPRL, PVS, Twelf, etc.), their use is still not commonplace.

We believe that the time is ripe to join the efforts of the two communities, bringing developers of automated proof assistants together with a large pool of eager potential clients—programming language designers and researchers. In particular, we would like to answer two questions:

1. What is the current state of the art in formalizing language metatheory and semantics? What can be recommended as best practices for groups (typically

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 50–65, 2005.

not proof-assistant experts) embarking on formalizing language definitions, either small- or large-scale?
2. What improvements are needed to make the use of tool support commonplace? What can each community contribute?

Over the past several months, we have surveyed the landscape of proof assistants, language representation strategies, and related tools. Collectively, we have applied automated theorem proving technology to a number of problems, including proving transitivity of the algorithmic subtype relation in System $F_{<:}$ [3], proving type soundness of Featherweight Java, proving type soundness of variants of the simply typed λ-calculus and $F_{<:}$, and a substantial formalization of the behavior of TCP, UDP, and the Sockets API. We have carried out these case studies using a variety of object-language representation strategies, proof techniques, and proving environments. We have also experimented with lightweight tools designed to make it easier to define and typeset both formal and informal mathematics. Although experts in programming language theory, we are relative outsiders with respect to computer-aided proof.

Our conclusion from these experiments is that the relevant technology has developed *almost* to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research—mechanized metatheory for the masses.

Tool support for formal reasoning about programming languages would be useful at many levels:

1. *Machine-checked metatheory.* These are the classic problems: type preservation and soundness theorems, unique decomposition properties for operational semantics, proofs of equivalence between algorithmic and declarative variants of type systems, etc. At present such results are typically proved by hand for small to medium-sized calculi, and are not proved at all for full language definitions. We envision a future in which the papers in conferences such as *Principles of Programming Languages (POPL)* are routinely accompanied by mechanically checkable proofs of the theorems they claim.
2. *Use of definitions as oracles for testing and animation.* When developing a language implementation together with a formal definition one would like to use the definition as an oracle for testing. This requires tools that can decide typing and evaluation relationships, and they might differ from the tools used for (1) or be embedded in the same proof assistant. In some cases one could use a definition directly as a prototype.
3. *Support for engineering large-scale definitions.* As we move to full language definitions—on the scale of Standard ML [19] or larger—pragmatic "software engineering" issues become increasingly important, as do the potential benefits of tool support. For large definitions, the need for elegant and concise notation becomes pressing, as witnessed by the care taken by present-day researchers using informal mathematics. Even lightweight tool support, without full mechanized proof, could be very useful in this domain, e.g., for sort

checking and typesetting of definitions and of informal proofs, automatically instantiating definitions, performing substitutions, etc.

Large scale formalization of languages is already within reach of current technology. For examples, see the work on proofs of correctness of the Damas-Milner type inference algorithm for ML [6,22], semantics for C [25], semantics for Standard ML [32,34,13], and semantics and proofs of correctness for substantial subsets of Java [24,17,23]. Some other significant existing applications of mechanized metatheory include Foundational Proof Carrying Code [1] and Typed Assembly Languages [4]. Inspired by these successes, we seek to make mechanized metatheory more accessible to programming languages researchers.

We hope to stimulate progress by providing a framework for comparing alternative technologies. We issue here an initial set of challenge problems, dubbed the PoplMark Challenge, chosen to exercise some aspects of programming languages that are known to be difficult to formalize: variable binding at both term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. Such challenge problems have been used in the past within the theorem proving community to focus attention on specific areas and to evaluate the relative merits of different tools; these have ranged in scale from benchmark suites and small problems [31,12,5,15,9,21] up to the grand challenges of Floyd, Hoare, and Moore [7,14,20]. We hope that our challenge will have a similarly stimulating effect.

Our problems are drawn from the basic metatheory of a call-by-value variant of System $F_{<:}$ [3], enriched with records, record subtyping, and record patterns. We provide an informal definition of its type system and operational semantics. This language is of moderate scale—significantly more complex than simply typed lambda-calculus or "mini ML," but much smaller than a full-blown programming language—to keep the work involved in attempting the challenges manageable. (Our challenges therefore explicitly address only points 1 and 2 above; we regard the pragmatic issues of point 3 as equally critical, but it is not yet clear to us how to formulate a useful challenge problem at this larger scale.) As these challenges are met, we anticipate that the set of problems will be extended to emphasize other language features and proof techniques.

We have begun collecting and publicizing solutions to these challenge problems on our web site.[1] In the longer run, we hope that this site and accompanying e-mail discussion list will serve as a forum for promoting and advancing the current best practices in proof assistant technology and making this technology available to the broader programming languages community and beyond. We encourage researchers to try out the PoplMark Challenge using their favorite tools and send us their solutions.

In the next section, we discuss in more detail our reasons for selecting this specific set of challenge problems. Section 3 describes the problems themselves, and Section 4 sketches some avenues for further development of the challenge problem set.

[1] `http://www.cis.upenn.edu/proj/plclub/mmm/`

2 Design of the Challenge

This section motivates our choice of challenge problems and discusses the evaluation criteria for proposed solutions to the challenges. Since variable binding is a central aspect of the challenges, we briefly discuss relevant techniques and sketch some of our own experience in this area.

2.1 Problem Selection

The goal of the PoplMark Challenge is to provide a small, well-defined set of problems that capture many of the most critical issues in formalizing programing language metatheory. By its nature, such a benchmark will not be able to reflect *all* important issues. Instead, the PoplMark problems concentrate on a few important features:

- *Binding.* Most programming languages have some form of binding in their syntax and require a treatment of α-equivalence and substitution in their semantics. To adequately represent many languages, the representation strategy must support multiple kinds of binders (e.g. term and type), constructs introducing multiple binders over the same scope, and potentially unbounded lists of binders (e.g. for record patterns).
- *Complex inductions.* Programming language definitions often involve complex, mutually recursive definitions. Structural induction over such objects, mutual induction, and induction on heights or pairs of derivations are all commonplace in metatheory.
- *Experimentation.* Proofs about programming languages are just one aspect of formalization; for some applications, experimenting with formalized language designs is equally interesting. It should be easy for the language designer to execute typechecking algorithms, generate sample program behaviors, and—most importantly—test real language implementations against the formalized definitions.
- *Component reuse.* To further facilitate experimentation with and sharing of language designs, the infrastructure should support some way of reusing prior definitions and parts of proofs.

We have carefully constructed the PoplMark Challenge to stress these features; a theorem-proving infrastructure that addresses the whole challenge should be applicable across a wide spectrum of programming language theory. While we believe that the features above are essential, our challenge does not address many other interesting and tricky-to-formalize constructs and reasoning principles. We discuss possible extensions to the challenge in Section 4.

2.2 Evaluation Criteria

A solution to the PoplMark Challenge will consist of appropriate software tools, a language representation strategy, and a demonstration that this infrastructure is sufficient to formalize the problems described in Section 3. The

long version of this paper (available at our web site) includes an appendix with reasonably detailed informal proofs of the challenge properties. Solutions to the challenge should follow the overall structure of these proofs, though we expect that details will vary from prover to prover and across term representations. In all cases, there must be an argument for why the formalization is equivalent to the presentation in Section 3—i.e., an adequacy theorem—which should be as simple as possible.

The primary metric of success (beyond correctness, of course) is that a solution should give us confidence of future success of other formalizations carried out using similar techniques. In particular, this implies that:

- *The technology should impose reasonable overheads.* We accept that there is a cost to formalization, and our goal is *not* to be able to prove things more easily than by hand (although that would certainly be welcome). We are willing to spend more time and effort to use the proof infrastructure, but the overhead of doing so must not be prohibitive. (For example, as we discuss below, our experience is that explicit de Bruijn-indexed representations of variable binding structure fail this test.)
- *The technology should be transparent.* The representation strategy and proof assistant syntax should not depart too radically from the usual conventions familiar to the technical audience, and the content of the theorems themselves should be apparent to someone not deeply familiar with the theorem proving technology used or the representation strategy chosen.
- *The technology should have a reasonable cost of entry.* The infrastructure should be usable (after, say, one semester of training) by someone who is knowledgeable about programming language theory but not an expert in theorem prover technology.

2.3 Representing Binders

The problem of representing and reasoning about inductively-defined structures with binders is central to the PoplMark challenges. Representing binders has been recognized as crucial by the theorem proving community, and many different solutions to this problem have been proposed. In our (still limited) experience, none emerge as clear winners. In this section we briefly summarize the main approaches and, where applicable, describe our own experiments using them. Our survey is far from complete and we refrain from drawing any hard conclusions, to give the proponents of each method a chance to try their hand at meeting the challenge.

A first-order, named approach very similar in flavor to standard informal presentations was used by Vestergaard and Brotherston to formalize some metatheory of untyped λ-calculus [35,36]. Their representation requires that each binder initially be assigned a unique name—one aspect of the so-called Barendregt convention.

Another popular concrete representation is de Bruijn's nameless representation. De Bruijn indices are easy to understand and support the full range of

induction principles needed to reason over terms. In our experience, however, de Bruijn representations have two major flaws. First, the *statements* of theorems require complicated clauses involving "shifted" terms and contexts. These extra clauses make it difficult to see the correspondence between informal and formal versions of the same theorem—there is no question of simply typesetting the formal statement and pasting it into a paper. Second, while the notational clutter is manageable for "toy" examples of the size of the simply-typed lambda calculus, we have found it becomes quite a heavy burden even for fairly small languages like $F_{<:}$.

In their formalization of properties of pure type systems, McKinna and Pollack use a hybrid approach that combines the above two representation strategies. In this approach, free variables are ordinary names while bound variables are represented using de Bruijn indices [18].

A radically different approach to representing terms with binders is higher-order abstract syntax (HOAS) [28]. In HOAS representations, binders in the meta-language are used to represent binders in the object language. Our experience with HOAS encodings (mainly as realized in Twelf) is that they provide a conveniently high level of abstraction, encapsulating much of the complexity of reasoning about binders. However, the strengths and limitations of the approach are not yet clearly understood, and it can sometimes require significant ingenuity to encode particular language features or proof ideas in this style.

Gordon and Melham propose a way to axiomatize inductive reasoning over untyped lambda-terms [11] and suggest that other inductive structures with binding can be encoded by setting up a correspondence with the untyped lambda terms. Norrish has pursued this direction [26,27], but observes that these axioms are cumbersome to use without some assistance from the theorem-proving tool. In particular, the axioms use universal quantification in inductive hypotheses where in informal practice "some/any" quantification is used. He has developed a library of lemmas about a system of permutations on top of the axioms that aids reasoning significantly.

Several recent approaches to binding take the concept of "swapping" as a primitive, and use it to build a nominal logic. Gabbay and Pitts proposed a method of reasoning about binders based upon a set theory extended with an intrinsic notion of permutation [8]. Pitts followed this up by proposing a new "nominal" logic based upon the idea of permutations [30]. More recent work by Urban proposes methods based on the same intuitions but carried out within a conventional logic [33]. Our own preliminary experiments with Urban's methods have been encouraging.

3 The Challenge

Our challenge problems are taken from the basic metatheory of System $F_{<:}$. This system is formed by enriching the types and terms of System F with a subtype relation, refining universal quantifiers to carry subtyping constraints, and adding records, record subtyping, and record patterns. Our presentation is based on Pierce's *Types and Programming Languages* [29].

The challenge comprises three distinct parts. The first deals just with the type language of $F_{<:}$; the second considers terms, evaluation, and type soundness. Each of these is further subdivided into two parts, starting with definitions and properties for *pure* $F_{<:}$ and then asking that the same properties be proved for $F_{<:}$ enriched with records and patterns. This partitioning allows the development to start small, but also—and more importantly—focuses attention on issues of reuse: How much of the first sub-part can be re-used verbatim in the second sub-part? The third problem asks that useful algorithms be extracted from the earlier formal definitions and used to "animate" some simple properties.

Challenge 1A: Transitivity of Subtyping

The first part of this challenge problem deals purely with the type language of $F_{<:}$. The syntax for this language is defined by the following grammar and inference rules. Although the grammar is simple—it has only four syntactic forms—some of its key properties require fairly sophisticated reasoning.

Syntax:

T ::=		*types*
	X	*type variable*
	Top	*maximum type*
	T→T	*type of functions*
	∀X<:T.T	*universal type*
Γ ::=		*type environments*
	∅	*empty type env.*
	Γ, X<:T	*type variable binding*

In $\forall \mathtt{X<:T_1}.\mathtt{T_2}$, the variable X is a binding occurrence with scope $\mathtt{T_2}$ (X is not bound in $\mathtt{T_1}$). In Γ, X<:T, the X must not be in the domain of Γ, and the free variables of T must all be in the domain of Γ.

Following standard practice, issues such as the use of α-conversion, capture avoidance during substitution, etc. are left implicit in what follows. There are several ways in which these issues can be formalized: we might take Γ as a concrete structure (such as an association list of named variables and types) but quotient types and terms up to alpha equivalence, or we could take entire judgments up to alpha equivalence. We might axiomatize the well-formedness of typing environments and types using auxiliary $\vdash \Gamma$ *ok* and $\Gamma \vdash \mathtt{T}$ *ok* judgments. And so on. We leave these decisions to the individual formalization.

It is acceptable to make small changes to the rules below to reflect these decisions, such as adding well-formedness premises. Changing the presentation of the rules to a notationally different but "obviously equivalent" style such as HOAS is also acceptable, but there must be a clear argument that it is really equivalent. Also, whatever formalization is chosen should make clear that we are only dealing with *well-scoped* terms. For example, it should not be possible to derive X <: Top in the empty typing environment.

The subtyping relation captures the intuition "if S is a subtype of T (written S <: T) then an instance of S may safely be used wherever an instance of T is expected." It is defined as the least relation closed under the following rules.

Subtyping $\boxed{\Gamma \vdash \mathtt{S} <: \mathtt{T}}$

$$\Gamma \vdash \mathtt{S} <: \mathtt{Top} \qquad \text{(SA-Top)}$$

$$\Gamma \vdash \mathtt{X} <: \mathtt{X} \qquad \text{(SA-Refl-TVar)}$$

$$\frac{\mathtt{X}<:\mathtt{U} \in \Gamma \qquad \Gamma \vdash \mathtt{U} <: \mathtt{T}}{\Gamma \vdash \mathtt{X} <: \mathtt{T}} \qquad \text{(SA-Trans-TVar)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 <: \mathtt{S}_1 \qquad \Gamma \vdash \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(SA-Arrow)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 <: \mathtt{S}_1 \qquad \Gamma, \mathtt{X}<:\mathtt{T}_1 \vdash \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash \forall \mathtt{X}<:\mathtt{S}_1 . \mathtt{S}_2 <: \forall \mathtt{X}<:\mathtt{T}_1 . \mathtt{T}_2} \qquad \text{(SA-All)}$$

These rules present an *algorithmic* version of the subtyping relation. In contrast to the more familiar *declarative* presentation, these rules are syntax-directed, as might be found in the implementation of a type checker; the algorithmic rules are also somewhat easier to reason with, having, for example, an obvious inversion property. The declarative rules differ from these by explicitly stating that subtyping is reflexive and transitive. However, reflexivity and transitivity also turn out to be derivable properties in the algorithmic system. A straightforward induction shows that the algorithmic rules are reflexive. The first challenge is to show that that they are also transitive.

3.1 Lemma [Transitivity of Algorithmic Subtyping]: If $\Gamma \vdash \mathtt{S} <: \mathtt{Q}$ and $\Gamma \vdash \mathtt{Q} <: \mathtt{T}$, then $\Gamma \vdash \mathtt{S} <: \mathtt{T}$. □

The difficulty here lies in the reasoning needed to prove this lemma. Transitivity must be proven simultaneously with another property, called *narrowing*, by an inductive argument with case analyses on the final rules used in the given derivations.

3.2 Lemma [Narrowing]: If Γ, $\mathtt{X}<:\mathtt{Q}$, $\Delta \vdash \mathtt{M} <: \mathtt{N}$ and $\Gamma \vdash \mathtt{P} <: \mathtt{Q}$ then $\Gamma, \mathtt{X}<:\mathtt{P}, \Delta \vdash \mathtt{M} <: \mathtt{N}$. □

Challenge 1B: Transitivity of Subtyping with Records

We now extend this challenge by enriching the type language with record types. The new syntax and subtyping rule for record types are shown below. Implicit in the syntax is the condition that the labels $\{\mathtt{l}_i \ ^{i \in 1..n}\}$ appearing in a record type $\{\mathtt{l}_i : \mathtt{T}_i \ ^{i \in 1..n}\}$ are pairwise distinct.

New syntactic forms:

T	::=	...	*types*
		$\{l_i : T_i \ ^{i \in 1..n}\}$	*type of records*

New subtyping rules $\boxed{\Gamma \vdash S <: T}$

$$\frac{\{l_i \ ^{i \in 1..n}\} \subseteq \{k_j \ ^{j \in 1..m}\} \qquad \text{if } k_j = l_i, \text{ then } \quad \Gamma \vdash S_j <: T_i}{\Gamma \vdash \{k_j : S_j \ ^{j \in 1..m}\} <: \{l_i : T_i \ ^{i \in 1..n}\}} \quad \text{(SA-RCD)}$$

Although it has been shown that records can actually be encoded in pure $F_{<:}$ [2,10], dealing with them directly is a worthwhile task since, unlike other syntactic forms, record types have an arbitrary (finite) number of fields. Also, the informal proof for Challenge 1A extends to record types by only adding the appropriate cases. A formal proof should reflect this.

Challenge 2A: Type Safety for Pure $F_{<:}$

The next challenge considers the type soundness of pure $F_{<:}$ (without record types, for the moment). Below, we complete the definition of $F_{<:}$ by describing the syntax of terms, values, and typing environments with term binders and giving inference rules for the typing relation and a small-step operational semantics.

As usual in informal presentations, we elide the formal definition of substitution and simply assume that the substitutions of a type P for X in T (denoted $[X \mapsto P]T$) and of a term q for x in t (denoted $[x \mapsto q]t$) are capture-avoiding.

Syntax:

t	::=		*terms*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
		λX<:T.t	*type abstraction*
		t [T]	*type application*
v	::=		*values*
		λx:T.t	*abstraction value*
		λX<:T.t	*type abstraction value*
Γ	::=		*type environments*
		$\emptyset$	*empty type env.*
		Γ, x:T	*term variable binding*
		Γ, X<:T	*type variable binding*

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x{:}T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x{:}T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{x{:}T_1}.\mathtt{t_2} : \mathtt{T_1} \to \mathtt{T_2}} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_{11}} \to \mathtt{T_{12}} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_{12}}} \quad \text{(T-App)}$$

$$\frac{\Gamma, \mathtt{X{<:}T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{X{<:}T_1}.\mathtt{t_2} : \forall \mathtt{X{<:}T_1}.\mathtt{T_2}} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \forall \mathtt{X{<:}T_{11}}.\mathtt{T_{12}} \qquad \Gamma \vdash \mathtt{T_2} <: \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1}\ [\mathtt{T_2}] : [\mathtt{X} \mapsto \mathtt{T_2}]\mathtt{T_{12}}} \quad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \quad \text{(T-Sub)}$$

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t'}}$

$$(\lambda \mathtt{x{:}T_{11}}.\mathtt{t_{12}})\ \mathtt{v_2} \longrightarrow [\mathtt{x} \mapsto \mathtt{v_2}]\mathtt{t_{12}} \quad \text{(E-AppAbs)}$$

$$(\lambda \mathtt{X{<:}T_{11}}.\mathtt{t_{12}})\ [\mathtt{T_2}] \longrightarrow [\mathtt{X} \mapsto \mathtt{T_2}]\mathtt{t_{12}} \quad \text{(E-TAppTAbs)}$$

Evaluation contexts:

E ::=	*evaluation contexts*
$[-]$	*hole*
$E\ \mathtt{t}$	*app fun*
$\mathtt{v}\ E$	*app arg*
$E\ [\mathtt{T}]$	*type fun*

Evaluation in context:

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t'_1}}{E[\mathtt{t_1}] \longrightarrow E[\mathtt{t'_1}]} \quad \text{(E-Ctx)}$$

The evaluation relation is presented in two parts: the rules E-AppAbs and E-TappTabs capture the immediate reduction rules of the language, while E-Ctx permits reduction under an arbitrary *evaluation context* E. For $F_{<:}$, one would have an equally clear definition and slightly simpler proofs using explicit closure rules for the evaluation relation. We use evaluation contexts with an

eye to larger languages and languages with non-local control operators such as exceptions, for which (in informal mathematics) they are an important tool for reducing notational clutter in definitions.[2] Evaluation contexts are also particularly interesting from the point of view of formalization when they include binders, though unfortunately there are no examples of this in call-by-value $F_{<:}$.

Type soundness is usually proven in terms of *preservation* and *progress* theorems. Challenge 2A is to prove these properties for pure $F_{<:}$.

3.3 Theorem [Preservation]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. □

3.4 Theorem [Progress]: If $\mathtt{t}$ is a closed, well-typed $F_{<:}$ term (i.e., if $\vdash \mathtt{t} : \mathtt{T}$ for some $\mathtt{T}$), then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

Unlike the proof of transitivity of subtyping, the inductive arguments required here are straightforward. However, variable binding becomes a more significant issue, since this language includes binding of both type and term variables. Several lemmas relating to both kinds of binding must also be shown, in particular lemmas about type and term substitutions. These lemmas, in turn, require reasoning about permuting, weakening, and strengthening typing environments.

Challenge 2B: Type Safety with Records and Pattern Matching

The next challenge is to extend the preservation and progress results to cover records and pattern matching. The new syntax and rules for this language appear below. As for record types, the labels $\{\mathtt{l}_i \ ^{i \in 1..n}\}$ appearing in a record $\{\mathtt{l}_i = \mathtt{t}_i \ ^{i \in 1..n}\}$ are assumed to be pairwise distinct. Similarly, the variable patterns appearing in a pattern are assumed to bind pairwise distinct variables.

New syntactic forms:

t	::=	...	*terms*
		$\{\mathtt{l}_i=\mathtt{t}_i \ ^{i \in 1..n}\}$	*record*
		t.l	*projection*
		let p=t in t	*pattern binding*
p	::=		*patterns*
		x:T	*variable pattern*
		$\{\mathtt{l}_i=\mathtt{p}_i \ ^{i \in 1..n}\}$	*record pattern*

[2] This design choice has generated a robust debate on the PoplMark discussion list as to whether evaluation contexts *must* be used in order for a solution to count as valid, or whether an "obviously equivalent" presentation such as an evaluation relation with additional congruence rules is acceptable. We prefer evaluation contexts for the reasons we have given, but the consensus of the community appears to be that one should accept solutions in other styles. However, a good solution must be formulated in a style that provides similar clarity as the language scales.

$$\begin{array}{lll} \mathtt{v} & ::= & \ldots \qquad\qquad \textit{values} \\ & & \{\mathtt{l}_i{=}\mathtt{v}_i\ ^{i\in 1..n}\} \qquad \textit{record value} \end{array}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \vdash \mathtt{p} : \mathtt{T}_1 \Rightarrow \Delta \qquad \Gamma, \Delta \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let\ p{=}t}_1\ \mathtt{in\ t}_2 : \mathtt{T}_2} \qquad \text{(T-Let)}$$

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i{=}\mathtt{t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{l}_j : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

Pattern typing rules:

$$\vdash (\mathtt{x{:}T}) : \mathtt{T} \Rightarrow \mathtt{x} : \mathtt{T} \qquad \text{(P-Var)}$$

$$\frac{\text{for each } i \quad \vdash \mathtt{p}_i : \mathtt{T}_i \Rightarrow \Delta_i}{\vdash \{\mathtt{l}_i{=}\mathtt{p}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\} \Rightarrow \Delta_n, \ \ldots, \Delta_1} \qquad \text{(P-Rcd)}$$

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{let\ p{=}v}_1\ \mathtt{in\ t}_2 \longrightarrow \mathit{match}(\mathtt{p}, \mathtt{v}_1)\mathtt{t}_2 \qquad \text{(E-LetV)}$$

$$\{\mathtt{l}_i{=}\mathtt{v}_i\ ^{i\in 1..n}\}.\mathtt{l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

New evaluation contexts:

$$\begin{array}{lll} E & ::= & \ldots \qquad\qquad \textit{evaluation contexts} \\ & & E.\mathtt{l} \qquad\qquad \textit{projection} \\ & & \{\mathtt{l}_i{=}\mathtt{v}_i\ ^{i\in 1..j-1}, \mathtt{l}_j{=}E, \mathtt{l}_k{=}\mathtt{t}_k\ ^{k\in j+1..n}\} \qquad \textit{record} \\ & & \mathtt{let\ p{=}}E\ \mathtt{in\ t}_2 \qquad\qquad \textit{let binding} \end{array}$$

Matching rules:

$$\mathit{match}(\mathtt{x{:}T}, \mathtt{v}) = [\mathtt{x} \mapsto \mathtt{v}] \qquad \text{(M-Var)}$$

$$\frac{\{\mathtt{l}_i\ ^{i\in 1..n}\} \subseteq \{\mathtt{k}_j\ ^{j\in 1..m}\} \qquad \text{if } \mathtt{l}_i = \mathtt{k}_j, \text{ then } \mathit{match}(\mathtt{p}_i, \mathtt{v}_j) = \sigma_i}{\mathit{match}(\{\mathtt{l}_i{=}\mathtt{p}_i\ ^{i\in 1..n}\}, \{\mathtt{k}_j{=}\mathtt{v}_j\ ^{j\in 1..m}\}) \quad = \quad \sigma_n \circ \cdots \circ \sigma_1} \qquad \text{(M-Rcd)}$$

Compared to the language of Challenge 2A, the `let` construct is a fundamentally new binding form, since patterns may bind an arbitrary (finite) number of term variables.

Challenge 3: Testing and Animating with Respect to the Semantics

Given a complete formal definition of a language, there are at least two interesting ways in which it can be used (as opposed to being reasoned about). When implementing the language, it should be possible to use the formal definition as an oracle for *testing* the implementation—checking that it does conform to the definition by running test cases in the implementation and confirming formally that the outcome is as prescribed. Secondly, one would like to construct a prototype implementation from the definition and use it for *animating* the language, i.e., exploring the language's properties on particular examples. In both cases, this should be done without any unverified (and thus error-prone) manual translation of definitions.

Our final challenge is to provide an implementation of this functionality, specifically for the following three tasks (using the language of Challenge 2B):

1. Given $F_{<:}$ terms $\mathtt{t}$ and $\mathtt{t}'$, decide whether $\mathtt{t} \longrightarrow \mathtt{t}'$.
2. Given $F_{<:}$ terms $\mathtt{t}$ and $\mathtt{t}'$, decide whether $\mathtt{t} \longrightarrow^* \mathtt{t}' \not\longrightarrow$, where $\longrightarrow^*$ is the reflexive-transitive closure of $\longrightarrow$.
3. Given an $F_{<:}$ term $\mathtt{t}$, find a term $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$.

The first two subtasks are useful for testing language implementations, while the last is useful for animating the definition. For all three subtasks, the system(s) should accept syntax that is "reasonably close" to that of informal (ASCII) mathematical notation, though it maybe necessary to translate between the syntaxes of a formal environment and an implementation. We will provide an implementation of an interpreter for $F_{<:}$ with records and patterns at the challenge's website in order to make this challenge concrete, together with a graded sequence of example terms. To make a rough performance comparison possible, solutions should indicate execution times for these terms.

A solution to this challenge might make use of decision procedures and tactics of a proof assistant or might extract stand-alone code. In general, it may be necessary to combine theorems (e.g. that a rule-based but algorithmic definition of typing coincides with a declarative definition) and proof search (e.g. deciding particular instances of the algorithmic definition).

4 Beyond the Challenge

The PoplMark Challenge is not meant to be exhaustive: other aspects of programming language theory raise formalization difficulties that are interestingly different from the problems we have proposed—to name a few: more complex binding constructs such as mutually recursive definitions, logical relations proofs, coinductive simulation arguments, undecidability results, and linear handling of type environments. As time goes on, we will issue a small number of further challenges highlighting the most important of these issues; suggestions from the community would be welcome. However, we believe that a technology that provides a good solution to the PoplMark challenge as we have formulated it

here will be sufficient to attract eager adopters in the programming languages community, beginning with the authors.

So what are you waiting for? It's time to bring mechanized metatheory to the masses!

Acknowledgments

A number of people have joined us in preliminary discussions of these challenge problems, including Andrew Appel, Karl Crary, Frank Pfenning, Bob Harper, Hugo Herbelin, Jason Hickey, Michael Norrish, Andrew Pitts, Randy Pollack, Carsten Schürmann, Phil Wadler, and Dan Wang. We are especially grateful to Randy Pollack for helping guide our earliest efforts at formalization and to Randy Pollack, Michael Norrish, and Carsten Schürmann for their own work on the challenge problems. Discussions on the PoplMark mailing list have greatly deepened our understanding of the current state of the art.

Sewell and Fairbairn acknowledge support from a Royal Society University Research Fellowship and EPSRC grant GR/T11715. Foster is supported by an NSF Graduate Research Fellowship. Weirich is supported in part by NSF grant 0347289 (*CAREER: Type-Directed Programming in Object-Oriented Languages*). Zdancewic is supported in part by NSF grant CCR-0311204 (*Dynamic Security Policies*) and NSF grant CNS-0346939 (*CAREER: Language-based Distributed System Security*).

References

1. Andrew W. Appel. Foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science (LICS), Boston, Massachusetts*, pages 247–258, June 2001.
2. Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC/Compaq Systems Research Center, January 1992. Also in C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994.
3. Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. Summary in TACS '91 (Sendai, Japan, pp. 750–770).
4. Karl Crary. Toward a foundational typed assembly language. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, pages 198–212, January 2003.
5. Louise A. Dennis. Inductive challenge problems, 2000. `http://www.cs.nott.ac.uk/~lad/research/challenges`.
6. Catherine Dubois and Valerie Menissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3-4):319–346, 1999.
7. Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

8. Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In *14th Symposium on Logic in Computer Science*, pages 214–224, 1999.
9. I. P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical Report APES-09-1999, APES, 1999. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).
10. Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Università di Pisa, 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.
11. Andrew D. Gordon and Tom Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer-Verlag, 1996.
12. Ian Green. The dream corpus of inductive conjectures, 1999. `http://dream.dai.ed.ac.uk/dc/lib.html`.
13. Elsa Gunter and Savi Maharaj. Studying the ML module system in HOL. *The Computer Journal: Special Issue on Theorem Proving in Higher Order Logics*, 38(2):142–151, 1995.
14. Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
15. Holger Hoos and Thomas Stuetzle. SATLIB. `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`.
16. J. J. Joyce and C.-J. H. Seger, editors. *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, Vancouver, B.C., Canada, August 1993. Springer-Verlag, 1994.
17. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.
18. James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4), November 1999.
19. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.
20. J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, Lisbon, Portugal*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2002.
21. J Strother Moore and George Porter. The apprentice challenge. *ACM Trans. Program. Lang. Syst.*, 24(3):193–216, 2002.
22. Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.
23. Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.
24. Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe—definitely. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–170, New York, NY, USA, 1998. ACM Press.

25. Michael Norrish. *Formalising C in HOL.* PhD thesis, Computer Laboratory, University of Cambridge, 1998.
26. Michael Norrish. Mechanising Hankin and Barendregt using the Gordon-Melham axioms. In *MERLIN '03: Proceedings Of The 2003 Workshop On Mechanized Reasoning About Languages With Variable Binding*, pages 1–7. ACM Press, 2003.
27. Michael Norrish. Recursive function definition for types with binders. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004*, volume 3223 of *Lecture Notes In Computer Science*, pages 241–256. Springer-Verlag, 2004.
28. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.
29. Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.
30. Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.
31. Geoff Sutcliffe and Christian Suttner. The TPTP problem library. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
32. Donald Syme. Reasoning with the formal definition of Standard ML in HOL. In Joyce and Seger [16], pages 43–60.
33. Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. Accepted at CADE-20 in Tallinn. See `http://www.mathematik.uni-muenchen.de/~urban/nominal/`.
34. Myra VanInwegen and Elsa Gunter. HOL-ML. In Joyce and Seger [16], pages 61–74.
35. René Vestergaard and James Brotherston. The mechanisation of Barendregt-style equational proofs (the residual perspective). In *Mechanized Reasoning about Languages with Variable Binding (MERLIN)*, volume 58 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.
36. René Vestergaard and James Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names. *Information and Computation*, 183(2):212 – 244, 2003. Special edition with selected papers from RTA01.

A Structured Set of Higher-Order Problems

Christoph E. Benzmüller and Chad E. Brown

Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany
www.ags.uni-sb.de/{~chris, ~cebrown}

Abstract. We present a set of problems that may support the development of calculi and theorem provers for classical higher-order logic. We propose to employ these test problems as quick and easy criteria preceding the formal soundness and completeness analysis of proof systems under development. Our set of problems is structured according to different technical issues and along different notions of semantics (including Henkin semantics) for higher-order logic. Many examples are either theorems or non-theorems depending on the choice of semantics. The examples can thus indicate the deductive strength of a proof system.

1 Motivation: Test Problems for Higher-Order Reasoning Systems

Test problems are important for the practical implementation of theorem provers as well as for the preceding theoretical development of calculi, strategies and heuristics. If the test theorems can be proven (resp. the non-theorems cannot) then they ideally provide a strong indication for completeness (resp. soundness). Examples for early publications providing first-order test problems are [21,29,23]. For more than decade now the TPTP library [28] has been developed as a systematically structured electronic repository of first-order test problems. This repository together with the yearly CASC theorem prover competitions [24] significantly supported the improvement of first-order and propositional reasoning systems. Unfortunately, a respective library of higher-order test problems is not yet available.

This paper presents a small set of significant test problems for classical higher-order logic that may guide the development of higher-order proof systems. These test problems are relevant for both automated and interactive higher-order theorem proving. Even some of our simpler theorems may be difficult to prove interactively. Examples are our problems 15(a): $p_{o\to o}\,(a_o \wedge b_o) \Rightarrow p\,(b \wedge a)$ and 16: $(p_{o\to o}\,a_o) \wedge (p\,b_o) \Rightarrow (p\,(a \wedge b))$.

Most of the examples presented here are chosen to be a simple representative of some particular technical or semantical point. We also include examples illustrating real challenges for higher-order theorem provers. Our work is relevant in the first place for theorem proving in classical higher-order logic. However, many of our examples also carry over to other logics such as intuitionistic higher-order logic. Most of the presented test problems evolved from experience gained in the development of the higher-order theorem provers TPS [5] and LEO [10,7]. Some of the examples and (many others) have been also discussed in other publications on classical higher-order logic, e.g. [15,17,6,1,4]. The novel contribution of this paper is not the test problems per se, but the connection of these examples with the particular model classes in which they are valid (resp. invalid) and their assemblage into a comprehensive set.

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 66–81, 2005.

We structure many of our examples along two dimensions. The examples are theorems or non-theorems depending on these dimensions.

Extensionality provides one dimension in which we can vary semantics. Assuming Henkin semantics, for instance, most of our examples denote theorems. If we choose a weaker semantics, for instance, by omitting Boolean extensionality, then some test problems become non-theorems providing a test case for soundness with respect to this more general notion of semantics (in which fewer propositions are valid). By varying extensionality, we have defined a landscape of eight higher-order model classes and developed abstract consistency methods and model existence results in [8,9]. This landscape of higher-order model classes and the corresponding abstract consistency framework provides much needed support for the theoretical analysis of the deductive power of calculi for higher-order logic. The test problems we introduce in this paper provide quick and easy test criteria for the soundness and completeness of proof systems with respect to these model classes. Testing a proof system with our examples should thus precede a formal, theoretical soundness and completeness analysis with the abstract consistency methodology introduced in [8,9].

Set comprehension provides another dimension along which one can vary semantics. In [14] different model classes are defined depending on the logical constants which occur in the signature. Since many sets are only definable in the presence of certain logical constants, this provides a way of varying the sets which exist in a model. In this paper, we provide examples of theorems which are only provable if one can use certain logical constants for instantiations. In implementations of the automated theorem provers TPS and LEO the problem of instantiating set variables corresponds to the use of *primitive substitutions* described in [14,2,3].

Section 2 introduces the syntax of classical higher-order logic following Church [15]. Section 3 presents some first test problems for pre-unification and quantifier dependencies. In Section 4 we review a landscape of higher-order semantics that distinguishes higher-order models with respect to various combinations of Boolean extensionality, three forms of functional extensionality and different signatures of logical constants. Section 5 provides test problems that are structured according to the introduced landscape of model classes. Section 6 presents some more complex test problems.

2 Classical Higher-Order Logic

As in [15], we formulate higher-order logic ($\mathcal{HOL}$) based on the simply typed λ-calculus. The set of simple types $\mathcal{T}$ is freely generated from basic types o and ι using the function type constructor $\to$.

For formulae we start with a set $\mathcal{V}$ of (typed) variables (denoted by $X_\alpha, Y, Z, \ldots$) and a signature Σ of (typed) constants (denoted by $c_\alpha, f_{\alpha\to\beta}, \ldots$). We let $\mathcal{V}_\alpha$ (Σ_α) denote the set of variables (constants) of type α. A signature Σ of constants may include logical constants from the set $\overline{\Sigma}$ defined by

$$\{\top_o, \bot_o, \neg_{o\to o}, \wedge_{o\to o\to o}, \vee_{o\to o\to o}, \Rightarrow_{o\to o\to o}, \Leftrightarrow_{o\to o\to o}\}$$

$$\cup\{\Pi^\alpha_{(\alpha\to o)\to o} \mid \alpha\in\mathcal{T}\}\cup\{\Sigma^\alpha_{(\alpha\to o)\to o} \mid \alpha\in\mathcal{T}\}\cup\{=^\alpha_{\alpha\to\alpha\to o} \mid \alpha\in\mathcal{T}\}.$$

Other constants in a signature are called parameters. The constants Π^α and Σ^α are used to define $\forall$ and $\exists$ (see below) without introducing a binding mechanism other than λ. The set of $\mathcal{HOL}$-formulae (or terms) over Σ are constructed from typed variables and constants using application and λ-abstraction. We let $\mathit{wff}_\alpha(\Sigma)$ be the set of all terms of type α and $\mathit{wff}(\Sigma)$ be the set of all terms. We use $\mathbf{A}, \mathbf{B}, \ldots$ to denote terms in $\mathit{wff}_\alpha(\Sigma)$.

We use vector notation to abbreviate k-fold applications and abstractions as $\mathbf{A}\overline{\mathbf{U}^k}$ and $\lambda\overline{X^k}.\mathbf{A}$, respectively. We also use Church's dot notation so that . stands for a (missing) left bracket whose mate is as far to the right as possible (consistent with given brackets). We use infix notation $\mathbf{A} \vee \mathbf{B}$ for $((\vee\mathbf{A})\mathbf{B})$ and binder notation $\forall X_\alpha.\mathbf{A}$ for $(\Pi^\alpha(\lambda X_\alpha.\mathbf{A}_o))$. While one can consider $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ to be defined (as in [8]), we consider these members of the signature Σ. We also use binder notation $\exists X.\mathbf{A}$ as shorthand for $\Sigma^\alpha(\lambda X.\mathbf{A})$ if Σ^α is a constant in Σ. We let $(\mathbf{A}_\alpha \doteq^\alpha \mathbf{B}_\alpha)$ denote the Leibniz equation $\forall P_{\alpha\to o}.(P\mathbf{A}) \Rightarrow .P\mathbf{B}$.

Each occurrence of a variable in a term is either free or bound by a λ. We use $free(\mathbf{A})$ to denote the set of free variables of $\mathbf{A}$ (i.e., variables with a free occurrence in $\mathbf{A}$). We consider two terms to be equal (written $\mathbf{A} \equiv \mathbf{B}$) if the terms are the same up to the names of bound variables (i.e., we consider α-conversion implicitly). A term $\mathbf{A}$ is closed if $free(\mathbf{A})$ is empty. We let $\mathit{cwff}_\alpha(\Sigma)$ denote the set of closed terms of type α and $\mathit{cwff}(\Sigma)$ denote the set of all closed terms. Each term $\mathbf{A} \in \mathit{wff}_o(\Sigma)$ is called a proposition and each term $\mathbf{A} \in \mathit{cwff}_o(\Sigma)$ is called a sentence.

We denote substitution of a term $\mathbf{A}_\alpha$ for a variable X_α in a term $\mathbf{B}_\beta$ by $[\mathbf{A}/X]\mathbf{B}$. Since we consider α-conversion implicitly, we assume the bound variables of $\mathbf{B}$ avoid variable capture.

Two common relations on terms are given by β-reduction and η-reduction. A β-redex $(\lambda X.\mathbf{A})\mathbf{B}$ β-reduces to $[\mathbf{B}/X]\mathbf{A}$. An η-redex $(\lambda X.\mathbf{C}X)$ (where $X \notin free(\mathbf{C})$) η-reduces to $\mathbf{C}$. For $\mathbf{A}, \mathbf{B} \in \mathit{wff}_\alpha(\Sigma)$, we write $\mathbf{A}{\equiv_\beta}\mathbf{B}$ to mean $\mathbf{A}$ can be converted to $\mathbf{B}$ by a series of β-reductions and expansions. Similarly, $\mathbf{A}{\equiv_{\beta\eta}}\mathbf{B}$ means $\mathbf{A}$ can be converted to $\mathbf{B}$ using both β and η. For each $\mathbf{A} \in \mathit{wff}(\Sigma)$ there is a unique β-normal form (denoted $\mathbf{A}{\downarrow_\beta}$) and a unique $\beta\eta$-normal form (denoted $\mathbf{A}{\downarrow_{\beta\eta}}$). From this fact we know $\mathbf{A}{\equiv_\beta}\mathbf{B}$ ($\mathbf{A}{\equiv_{\beta\eta}}\mathbf{B}$) iff $\mathbf{A}{\downarrow_\beta} \equiv \mathbf{B}{\downarrow_\beta}$ ($\mathbf{A}{\downarrow_{\beta\eta}} \equiv \mathbf{B}{\downarrow_{\beta\eta}}$).

A non-atomic formula in $\mathit{wff}_o(\Sigma)$ is any formula whose β-normal form is $(c\overline{\mathbf{A}^n})$ where c is a logical constant. An atomic formula is any other formula in $\mathit{wff}_o(\Sigma)$.

Many of the example problems in this paper employ equality, e.g. $\neg(a = \neg a)$. We have different options for the encoding of equality. We can either use primitive equality (i.e., equality as a logical constant) or use some definition of equality in terms of other logical constants. A common definition is Leibniz equality $(\forall P_{\alpha\to o}.(P\mathbf{A}) \Rightarrow .P\mathbf{B})$, but others are possible (see Exercise **X5303** in [4]). In many examples we will denote equality by $\stackrel{*}{=}$ (e.g., $\neg(a \stackrel{*}{=} \neg a)$). For each different interpretation of equality, we obtain a different example. We will discuss conditions under which different choices lead to theorems and which choices lead to non-theorems.

For some types, one can also define equality extensionally. For example, one can use equivalence instead of equality at type o. Similarly, at any type $\alpha \to o$, we introduce $\stackrel{set}{=}$ to denote set equality, i.e., $\stackrel{set}{=}$ is an abbreviation for

$$\lambda U_{\alpha\to o}\lambda V_{\alpha\to o}\forall X_\alpha.UX \Leftrightarrow VX.$$

In some cases, the use of an extensional definition of equality yields a theorem which can be proven without *assuming* extensionality. We will *not* use the notation $\stackrel{*}{=}$ to refer to any extensional definition of equality. Interpreting $\stackrel{*}{=}$ extensionally would significantly change some of the discussion below.

3 Test Problems for Pre-unification and Quantifier Dependencies

Higher-order pre-unification (see [26]) and higher-order Skolemization (see [22]) are important basic ingredients for building an automated higher-order theorem prover. They are largely independent of the chosen semantics for higher-order logic with one exception: β versus $\beta\eta$. As noted in [18] the unification problem relative to β-conversion is different from the unification problem relative to $\beta\eta$-conversion.

3.1 Pre-unification

Implementing a sound, complete and efficient pre-unification algorithm for the simply typed λ-calculus is a highly non-trivial task. Since higher-order pre-unification extends standard first-order unification all first-order test problems in the literature also apply to the higher-order case.

Some specific higher-order test problems can be obtained from the literature on higher-order unification and pre-unification, for example [26,25]. We will now illustrate how further challenging test examples can be easily created using Church numerals.

Church numerals are usually employed in the context of the untyped λ-calculus to encode the natural numbers. This encoding can be partly transformed in a simply typed or polymorphic typed λ-calculus. This includes the definition of successor, addition and multiplication which we employ in or test problems.

Iteration is the key concept to encode natural numbers as Church numerals. For each type α, we can define the Church numeral $\overline{n}^{\alpha}$ by $(\lambda F_{\alpha\to\alpha}\lambda Y_{\alpha}.(F^n Y))_{(\alpha\to\alpha)\to(\alpha\to\alpha)}$ where $(F^n Y)$ is shorthand for $\underbrace{(F\ (F \ldots (F}_{n-times}\ Y)))$. We will often write $\overline{n}$ instead of $\overline{n}^{\alpha}$, leaving the dependence on the type implicit. Omitting types[1], the successor function $\overline{s}$ can be defined as $\lambda N\lambda F\lambda Y.F(NFY)$, addition $\overline{+}$ as $\lambda M\lambda N\lambda F\lambda Y.MF(NFY)$ and multiplication $\overline{\times}$ as $\lambda M\lambda N\lambda F\lambda Z.N(MF)Z$. To ease notation, we write $\overline{+}$ and $\overline{\times}$ in infix.

Arithmetic equations on Church numerals such as $\overline{3}\overline{\times}\overline{4}\stackrel{*}{=}\overline{5}\overline{+}\overline{7}$ or $(((\overline{1}\overline{0}\overline{\times}\overline{1}\overline{0})\overline{\times}\overline{1}\overline{0})\stackrel{*}{=}((\overline{1}\overline{0}\overline{\times}\overline{5})\overline{+}(\overline{5}\overline{\times}\overline{1}\overline{0}))\overline{\times}\overline{1}\overline{0}))$ provide highly suited test problems for the efficiency of β-conversion or $\beta\eta$-conversion in the proof system. Of course, in order to correctly implement β- and η-conversion, one must first properly implement α-conversion.

We obtain more challenging test problems if we employ pre-unification for synthesizing Church numerals and arithmetical operations.

Example 1. (Solving arithmetical equations using pre-unification) The following examples are provable using pre-unification for β-conversion.

[1] N, M are of type $(\alpha\to\alpha)\to(\alpha\to\alpha)$, F is of type $\alpha\to\alpha$, and Y, Z are of type α.

(a) $\exists N_{(\iota\to\iota)\to\iota\to\iota}.((N\overline{\times 1}) \stackrel{*}{=} \overline{1})$ (There are two solutions, $\overline{1}$ and $(\lambda F_{\iota\,\iota}.F)$, if one only assumes β-conversion. There is one solution assuming $\beta\eta$-conversion.)
(b) $\exists N.(N\overline{\times 4}) \stackrel{*}{=} \overline{5+7}$
(c) $\exists H.(((H\bar{2})\bar{3}) \stackrel{*}{=} \bar{6}) \wedge (((H\bar{1})\bar{2}) \stackrel{*}{=} \bar{2}))$
(d) $\exists N, M.(N\overline{\times 4}) \stackrel{*}{=} \overline{5+}M$ (There are infinitely many solutions to this problem.)

3.2 Quantifier Dependencies

In proof search with tableaux and expansion proofs, variable conditions can be used to encode quantifier dependencies. Of course, one must be careful to obtain a sound framework. For instance, the variable conditions added with each eliminated existential quantifier in the framework used in [20] allow (incorrect) proofs of the following first-order non-theorems:

Example 2. (First-order non-theorems)

(a) (Example 2.9 in [30]) $(\exists X_\iota \forall Y_\iota . q_{\iota\to\iota\to o} XY) \vee (\exists U_\iota \forall V_\iota . \neg qVU)$
(b) (Example 2.50 in [30]) $\exists Y_\iota \forall X_\iota . ((\forall Z_\iota . q_{\iota\to\iota\to o} XZ) \vee (\neg qXY))$

In [19] an attempt was made to use variable conditions in the context of resolution theorem proving (for a sorted extension of higher-order logic) instead of introducing Skolem terms. However, the system was unsound as it allowed a resolution refutation proving the following non-theorem:

Example 3. (Non-Theorem: Every function has a fixed point) $\forall F_{\alpha\to\alpha}.\exists X_\alpha.F\ X \doteq X$. The idea is that one obtains two single-literal clauses $(P_{\iota\to o}(FX))$ and $\neg(PY)$ using clause normalization and variable renaming (where X and Y can be instantiated). One then obtains the empty clause by unifying Y with (FX).

Skolem terms avoid incorrect proofs of such theorems since the Skolem terms will preserve the relationship between renamed variables in different clauses. In particular, if S is a Skolem function, we would obtain single-literal clauses $(S_{\iota\to\iota\to o} X(FX))$ and $\neg(S_{\iota\to\iota\to o} YY)$ which cannot be resolved and unified.

There is a relationship between Skolemization and the axiom of choice in the first-order case which becomes more delicate in the higher-order case. Consider formulas $\forall x_\iota \exists y_\iota \varphi(x,y)$ and $\forall x_\iota \varphi(x,(f_{\iota\to\iota} x))$. In first-order logic, the two formulas are equivalent with respect to satisfiability whenever f does not occur in φ. The equivalence follows from the fact that any first-order model (with domain $\mathcal{D}_\iota$) satisfying $\forall x \exists y \varphi(x,y)$ can be extended to interpret f as a function $g : \mathcal{D}_\iota \longrightarrow \mathcal{D}_\iota$ such that $\forall x \varphi(x,(fx))$ holds. In general, the axiom of choice (at the *meta-level*) is required to conclude the function g exists. The situation is different in the higher-order case. As we shall see when we consider higher-order models, we would need to interpret f not simply as a function from $\mathcal{D}_\iota$ to $\mathcal{D}_\iota$, but as a member of a domain $\mathcal{D}_{\iota\to\iota}$. Existence of an appropriate function from $\mathcal{D}_\iota$ to $\mathcal{D}_\iota$ follows from the axiom of choice at the meta-level, but the existence of an appropriate element of $\mathcal{D}_{\iota\to\iota}$ would only follow from a choice property internal to the higher-order model.

Dale Miller has shown that a naive adaptation of standard first-order Skolemization to higher-order logic allows one to prove particular instances of the axiom of choice.

For example, naive Skolemization permits an easy proof of the following version of the axiom of choice:

Example 4. (Choice) $(\forall X \exists Y . rXY) \Rightarrow (\exists F \forall X . rX(FX))$

However, naive Skolemization does not provide a complete method for reasoning with choice. The following example is equivalent to the axiom of choice (essentially Axiom 11 in [15]) but is not provable using naive Skolemization.

Example 5. (Choice) $\exists E_{(\iota\to o)\to\iota} \forall P . (\exists Y . PY) \Rightarrow . P(EP)$

Thus standard first-order Skolemization is unsound in higher-order logic as it partly introduces choice into the proof system. Dale Miller has fixed the problem by adding further conditions (see [22]): any Skolem function symbol f^n with dependency arity n (the existentially bound variable to be eliminated by a new Skolem term headed by f is depending on n universial variables) may only occur in formulas $f^n\overline{\mathbf{A}^n}$, where none of the $\mathbf{A}^i$ contains a variable that is bound outside of the term $f^n\overline{\mathbf{A}^n}$.

4 Semantics for HOL

In [8] we have re-examined the semantics of classical higher-order logic with the purpose of clarifying the role of extensionality. For this we have defined eight classes of higher-order models with respect to various combinations of Boolean extensionality and three forms of functional extensionality. One can further refine these eight model classes by varying the logical constants in the signature Σ as in [14].

A model of $\mathcal{HOL}$ is given by four objects: a typed collection of nonempty sets $(\mathcal{D}_\alpha)_{\alpha\in\mathcal{T}}$, an application operator $@\colon \mathcal{D}_{\alpha\to\beta}\times\mathcal{D}_\alpha \longrightarrow \mathcal{D}_\beta$, an evaluation function $\mathcal{E}$ for terms and a valuation function $v\colon \mathcal{D}_o \longrightarrow \{\mathtt{T},\mathtt{F}\}$. A pair $(\mathcal{D},@)$ is called a Σ-applicative structure (see [8](3.1)). If $\mathcal{E}$ is an evaluation function for $(\mathcal{D},@)$ (see [8](3.18)), then we call the triple $(\mathcal{D},@,\mathcal{E})$ a Σ-evaluation. If v satisfies appropriate properties, then we call the tuple $(\mathcal{D},@,\mathcal{E},v)$ a Σ-model (see [8](3.40 and 3.41)).

Given an applicative structure $(\mathcal{D},@)$, an assignment φ is a (typed) function from $\mathcal{V}$ to $\mathcal{D}$. An evaluation function $\mathcal{E}$ maps an assignment φ and a term $\mathbf{A}_\alpha \in \mathit{wff}_\alpha(\Sigma)$ to an element $\mathcal{E}_\varphi(\mathbf{A}) \in \mathcal{D}_\alpha$. Evaluation functions $\mathcal{E}$ are required to satisfy four properties given in [8](3.18)). If $\mathbf{A}$ is closed and $\mathcal{E}$ is an evaluation function, then $\mathcal{E}_\varphi(\mathbf{A})$ cannot depend on φ and we write $\mathcal{E}(\mathbf{A})$.

A valuation $v\colon \mathcal{D}_o \longrightarrow \{\mathtt{T},\mathtt{F}\}$ is required to satisfy a property $\mathfrak{L}_c(\mathcal{E}(c))$ for every logical constant $c \in \Sigma$ (see [8](3.40)). For each logical constant c, $\mathfrak{L}_c(\mathsf{a})$ is defined to hold if a is an object of a domain $\mathcal{D}_\alpha$ satisfying the characterizing property of the logical constant c. For example, $\mathfrak{L}_\neg(\mathsf{n})$ holds for $\mathsf{n} \in \mathcal{D}_{o\to o}$ iff for every $\mathsf{a} \in \mathcal{D}_o$, $v(\mathsf{n}@\mathsf{a})$ is $\mathtt{T}$ iff $v(\mathsf{a})$ is $\mathtt{F}$. Likewise, $\mathfrak{L}_{=^\alpha}(\mathsf{q})$ holds for $\mathsf{q} \in \mathcal{D}_{\alpha\to\alpha\to o}$ if for every $\mathsf{a},\mathsf{b} \in \mathcal{D}_\alpha$, $v(\mathsf{q}@\mathsf{a}@\mathsf{b})$ is $\mathtt{T}$ iff a equals b.

Given a model $\mathcal{M} := (\mathcal{D},@,\mathcal{E},v)$, an assignment φ and a proposition $\mathbf{A}$ (or set of propositions Φ), we say $\mathcal{M}$ satisfies $\mathbf{A}$ (or Φ) and write $\mathcal{M} \models_\varphi \mathbf{A}$ (or $\mathcal{M} \models_\varphi \Phi$) if $v(\mathcal{E}_\varphi(\mathbf{A})) \equiv \mathtt{T}$ (or $v(\mathcal{E}_\varphi(\mathbf{A})) \equiv \mathtt{T}$ for each $\mathbf{A} \in \Phi$). If $\mathbf{A}$ is closed (or every member of Φ is closed), then we simply write $\mathcal{M} \models \mathbf{A}$ (or $\mathcal{M} \models \Phi$) and say $\mathcal{M}$ is a model of $\mathbf{A}$ (or Φ). We also consider classes $\mathfrak{M}$ of Σ-models and say a proposition $\mathbf{A}$ is valid in $\mathfrak{M}$ if $\mathcal{M} \models_\varphi \mathbf{A}$ for every $\mathcal{M} \in \mathfrak{M}$ and assignment φ.

In order to define model classes which correspond to different notions of extensionality, we define five properties of models (see [8](3.46, 3.21 and 3.5)). For each Σ-model $\mathcal{M} := (\mathcal{D}, @, \mathcal{E}, \upsilon)$, we say $\mathcal{M}$ satisfies property

$\mathfrak{q}$ iff for all $\alpha \in \mathcal{T}$ there is a $\mathsf{q}^\alpha \in \mathcal{D}_{\alpha\to\alpha\to o}$ with $\mathfrak{L}_{=^\alpha}(\mathsf{q}^\alpha)$.
η iff $(\mathcal{D}, @, \mathcal{E})$ is η-functional (i.e., for each $\mathbf{A} \in \mathit{wff}_\alpha(\Sigma)$ and assignment φ, $\mathcal{E}_\varphi(\mathbf{A}) \equiv \mathcal{E}_\varphi(\mathbf{A}\downarrow_{\beta\eta})$).
ξ iff $(\mathcal{D}, @, \mathcal{E})$ is ξ-functional (i.e., for each $\mathbf{M}, \mathbf{N} \in \mathit{wff}_\beta(\Sigma)$, $X \in \mathcal{V}_\alpha$ and assignment φ, $\mathcal{E}_\varphi(\lambda X_\alpha.\mathbf{M}_\beta) \equiv \mathcal{E}_\varphi(\lambda X_\alpha.\mathbf{N}_\beta)$ whenever $\mathcal{E}_{\varphi,[\mathsf{a}/X]}(\mathbf{M}) \equiv \mathcal{E}_{\varphi,[\mathsf{a}/X]}(\mathbf{N})$ for every $\mathsf{a} \in \mathcal{D}_\alpha$).
$\mathfrak{f}$ iff $(\mathcal{D}, @)$ is functional (i.e., for each $\mathsf{f}, \mathsf{g} \in \mathcal{D}_{\alpha\to\beta}$, $\mathsf{f} \equiv \mathsf{g}$ whenever $\mathsf{f}@\mathsf{a} \equiv \mathsf{g}@\mathsf{a}$ for every $\mathsf{a} \in \mathcal{D}_\alpha$).
$\mathfrak{b}$ iff υ is injective.

For each $* \in \{\beta, \beta\eta, \beta\xi, \beta\mathfrak{f}, \beta\mathfrak{b}, \beta\eta\mathfrak{b}, \beta\xi\mathfrak{b}, \beta\mathfrak{f}\mathfrak{b}\}$ and each signature Σ we define $\mathfrak{M}_*(\Sigma)$ to be the class of all Σ-models $\mathcal{M}$ such that $\mathcal{M}$ satisfies property $\mathfrak{q}$ and each of the additional properties $\{\eta, \xi, \mathfrak{f}, \mathfrak{b}\}$ indicated in the subscript $*$ (see [8](3.49)). We always include β in the subscript to indicate that β-equal terms are always interpreted as identical elements. We do not include property $\mathfrak{q}$ as an explicit subscript; $\mathfrak{q}$ is treated as a basic, implicit requirement for all model classes. See [8](3.52) for a discussion on why we require property $\mathfrak{q}$. (We also briefly explore models which do not satisfy property $\mathfrak{q}$ in the context of Example 8 and again in Subsection 5.3.) Since we are varying four properties, one would expect to obtain 16 model classes. However, we showed in [8] that $\mathfrak{f}$ is equivalent to the conjunction of ξ and η. Note that, for example, $\mathfrak{M}_{\beta\mathfrak{f}}(\Sigma)$ is a larger class of models than $\mathfrak{M}_{\beta\mathfrak{f}\mathfrak{b}}(\Sigma)$, hence fewer propositions are valid in $\mathfrak{M}_{\beta\mathfrak{f}}(\Sigma)$ than are valid in $\mathfrak{M}_{\beta\mathfrak{f}\mathfrak{b}}(\Sigma)$. In our examples we try to indicate the largest of our model classes in which the proposition is valid. Implicitly, this means the proposition is also valid in smaller (more restricted) model classes and may not be valid in larger (less restricted) ones.

5 Test Problems for Higher-Order Theories

Unless stated otherwise, we assume the signature includes $\overline{\Sigma}$ (see p. 67) and write $\mathfrak{M}_*$ for $\mathfrak{M}_*(\Sigma)$. Many of the examples could be considered in the context of smaller signatures. In the following discussion, we only consider smaller signatures in order to make particular points. (Note that if the signature becomes too small, Leibniz equality, for example, is no longer expressible.)

5.1 Properties of Equality

There are many useful first-order test problems on equality reasoning in the literature. For instance, in [12] the following clause set is given to illustrate the incompleteness of the RUE-NRF resolution approach as introduced in [16]:

$$\{g(f(a)) = a, f(g(X)) \neq X\}$$

Here, X is a free variable (i.e., implicitly universially quantified) and f, g are unary function symbols. In [12] it is shown that this inconsistent clause set cannot be refuted in the first-order RUE-NRF approach.

We now present some higher-order test problems addressing properties of equality. Some of them apply to many possible notions of equality while others describe specific properties of individual notions or relate different notions to each other.

Example 6. Equality is an equivalence relation in $\mathfrak{M}_\beta$. These particular examples should be theorems even if one replaces $\stackrel{*}{=}$ with an extensional definition of equality (e.g., $\Leftrightarrow$ at type o or $\stackrel{set}{=}$ at any type $\alpha \to o$).

(a) $\forall X_\alpha . X \stackrel{*}{=} X$
(b) $\forall X_\alpha \forall Y_\alpha . X \stackrel{*}{=} Y \Rightarrow Y \stackrel{*}{=} X$
(c) $\forall X_\alpha \forall Y_\alpha \forall Z_\alpha . (X \stackrel{*}{=} Y \wedge Y \stackrel{*}{=} Z) \Rightarrow X \stackrel{*}{=} Z$

Example 7. Equality obeys the congruence property (substitutivity property) in $\mathfrak{M}_\beta$.

(a) $\forall X_\alpha \forall Y_\alpha \forall F_{\alpha\to\alpha} . X \stackrel{*}{=} Y \Rightarrow (FX) \stackrel{*}{=} (FY)$
(b) $\forall X_\alpha \forall Y_\alpha \forall P_{\alpha\to o} . (X \stackrel{*}{=} Y) \wedge (PX) \Rightarrow (PY)$

Example 8 relates the Leibniz definition of equality to primitive equality.

Example 8. $(a_\alpha \doteq^\alpha b_\alpha) \Rightarrow (a =^\alpha b)$.

One could legitimately debate whether Example 8 should be a theorem. On the one hand, if Example 8 is not a theorem, then one should not consider Leibniz equality to be a definition of *real* equality. Semantically, Henkin's first (quite natural) definitions of models allowed models in which Leibniz equality (e.g., at type ι) does *not* evaluate to equality of objects in the model. Such a model $\mathcal{M}$ is constructed in [1]. This model $\mathcal{M}$ is a Σ-model in the sense of this paper (if one assumes $=^\alpha \notin \Sigma$ for every type α), but is not in any model class $\mathfrak{M}_*(\Sigma)$ since property $\mathfrak{q}$ fails. There is a slight technical problem with saying $\mathcal{M}$ provides a counter-model for Example 8 since one cannot express Example 8 without $=^\iota \in \Sigma$. As in [14], one can distinguish between *internal* and *external* uses of equality (as well as $\Rightarrow$ and $\forall$) and determine that $\mathcal{M}$ *is* (in a sense that can be made precise) a countermodel for Example 8.

If a model satisfies property $\mathfrak{q}$, then Example 8 is valid for any type α. If a logical system is intended to be complete for one of our model classes $\mathfrak{M}_*(\Sigma)$, then Example 8 should be a theorem. For the complete natural deduction calculi in [8], there is an explicit rule which derives primitive equality from Leibniz equality. In some sense, requiring property $\mathfrak{q}$ semantically corresponds to explicitly requiring that Example 8 be provable.

Also, if $=^\alpha \in \Sigma$, then Example 8 (for this particular type α) is valid in any Σ-model. A proof *using* primitive equality could instantiate the Leibniz variable $P_{\alpha\to o}$ with $(\lambda Z_\alpha . a = Z)$. The important point is that $=$ must be available for instantiations during proofs (not simply for expressing the original sentence).

Extensionality is the distinguishing property motivating our different model classes. For both, functional and Boolean extensionality, we distinguish between a trivial and a non-trivial direction.

Example 9. The trivial directions of functional and Boolean extensionality are valid in $\mathfrak{M}_\beta$.

(a) $\forall F_{\alpha\to\beta}\forall G_{\alpha\to\beta}.F \stackrel{*}{=} G \Rightarrow (\forall X_\alpha.(FX) \stackrel{*}{=} (GX))$
(b) $\forall A_o\forall B_o.A \stackrel{*}{=} B \Rightarrow (A \Leftrightarrow B)$

The other directions are not valid in $\mathfrak{M}_\beta$. They become theorems only relative to more restricted model classes in our landscape.

Example 10. (discussed in [15]; Axiom 10 in [17]) $\forall A_o\forall B_o.(A \Leftrightarrow B) \Rightarrow A \stackrel{*}{=} B$ is valid in $\mathfrak{M}_{\beta\mathfrak{b}}$. This is the non-trivial direction of Boolean extensionality.

Example 11. ([15,17], Axiom $10^{\beta\alpha}$) $\forall F_{\alpha\to\beta}\forall G_{\alpha\to\beta}.(\forall X_\alpha.(FX) \stackrel{*}{=} (GX)) \Rightarrow F \stackrel{*}{=} G$ is valid in $\mathfrak{M}_{\beta\mathfrak{f}}$. This is the non-trivial direction of functional extensionality. (Property $\mathfrak{q}$ is also relevant to this example as is discussed in [8].)

5.2 Extensionality

We next present examples that illustrate distinguishing properties of the different model classes with respect to extensionality. In the preceding sections we have already mentioned several test problems that are independent of the "amount of extensionality" and which are theorems in $\mathfrak{M}_\beta$. We additionally refer to all first-order test problems as, for instance, provided in the TPTP library.

η-equality is usually realized as part of the pre-unification algorithm in a higher-order reasoning system. It is important to note that η-equality should not be confused with full extensionality. In literature on higher-order rewriting, for instance [25], the notion of extensionality is usually only associated with η-conversion which is far less than full extensionality.

Example 12. $(p_{(\iota\to\iota)\to o}(\lambda X_\iota.f_{\iota\to\iota}X)) \Rightarrow (p_{(\iota\to\iota)\to o}f)$ is essentially 21 from [15] which expresses η-equality using Leibniz equality. It is valid in $\mathfrak{M}_{\beta\eta}$ but not in $\mathfrak{M}_\beta$.

Property ξ together with η gives us full functional extensionality.

Example 13. Validity of $(\forall X_\iota.(f_{\iota\to\iota}X) \stackrel{*}{=} X) \wedge p(\lambda X_\iota X) \Rightarrow p(\lambda X_\iota.fX)$ only depends on ξ, not on η. It is thus valid in $\mathfrak{M}_{\beta\xi}$ (but not in model classes which do not require either ξ or $\mathfrak{f}$).

Example 14. $(\forall X_\iota.(f_{\iota\to\iota}X) \stackrel{*}{=} X) \wedge p(\lambda X_\iota X) \Rightarrow pf$ is valid in $\mathfrak{M}_{\beta\mathfrak{f}}$, but not in model classes which do not require $\mathfrak{f}$.

As in Example 11, property $\mathfrak{q}$ is important for validity of Example 13 in $\mathfrak{M}_{\beta\xi}$ and validity of Example 14 in $\mathfrak{M}_{\beta\mathfrak{f}}$.

Example 15. ([7]) (a) $p_{o\to o}\ (a_o \wedge b_o) \Rightarrow p\ (b \wedge a)$ and (b) $a_o \wedge b_o \wedge (p_{o\to o}a) \Rightarrow (pb)$ are valid iff we require Boolean extensionality as in $\mathfrak{M}_{\beta\mathfrak{b}}$.

Example 16. $(p_{o\to o}\ a_o) \wedge (p\ b_o) \Rightarrow (p\ (a \wedge b))$ is a theorem of $\mathfrak{M}_{\beta\mathfrak{b}}$ which is slightly more complicated to mechanize in some calculi; see [7] for more details.

Example 17. $\neg(a = \neg a)$ is valid in $\mathfrak{M}_{\beta\mathfrak{b}}$. As discussed in [7] this example motivates specific inference rules for the mechanization of primitive equality.

The following is a tricky example introduced in [14].

Example 18. $(h_{o\to\iota}((h\top) \stackrel{*}{=} (h\bot))) \stackrel{*}{=} (h\bot)$ is valid in $\mathfrak{M}_{\beta\mathfrak{b}}$, but not in model classes which do not require property $\mathfrak{b}$.

Many people do not immediately accept that Example 18 is a theorem. A simple informal argument is helpful. Either $(h\top) \stackrel{*}{=} (h\bot)$ is true or false. If the equation holds, then Example 18 reduces to $(h\top) \stackrel{*}{=} (h\bot)$ which we have just assumed. If the equation is false, then Example 18 reduces to $(h\bot) \stackrel{*}{=} (h\bot)$, an instance of reflexivity.

Example 19 combines Boolean extensionality with η-equality.

Example 19. $p_{(\iota\to\iota)\to o}(\lambda X_\iota . f_{o\to\iota\to\iota}(a_{(\iota\to\iota)\to o}(\lambda X_\iota . fb_o X)\wedge b)X) \Rightarrow p(f(b\wedge . a(fb)))$ is valid in $\mathfrak{M}_{\beta\eta\mathfrak{b}}$, but is not valid if properties $\mathfrak{b}$ and η are not assumed.

By DeMorgan's Law, we know $X \wedge Y$ is the same as $\neg(\neg X \vee \neg Y)$. In Example 20, we vary the notion of "is the same as" to obtain several examples which are only provable with some amount of extensionality. Note that if we only assume property ξ, we can only conclude the η-expanded form of $\wedge$ is equal to $(\lambda X \lambda Y . \neg(\neg X \vee \neg Y))$.

Example 20. Consider the following examples.

(a) $\forall X \forall Y . X \wedge Y \Leftrightarrow \neg(\neg X \vee \neg Y)$ is valid in $\mathfrak{M}_{\beta}$.
(b) $\forall X \forall Y . X \wedge Y \stackrel{*}{=} \neg(\neg X \vee \neg Y)$ is valid in $\mathfrak{M}_{\beta\mathfrak{b}}$.
(c) $(\lambda U \lambda V . U \wedge V) \stackrel{*}{=} (\lambda X \lambda Y . \neg(\neg X \vee \neg Y))$ is valid in $\mathfrak{M}_{\beta\xi\mathfrak{b}}$.
(d) $\wedge \stackrel{*}{=} (\lambda X \lambda Y . \neg(\neg X \vee \neg Y))$ is valid in $\mathfrak{M}_{\beta\mathfrak{f}\mathfrak{b}}$.

Finally we reach Henkin semantics which is characterized by full extensionality, i.e. the combination of Boolean and functional extensionality. Example 20(d) already provided one example valid only in $\mathfrak{M}_{\beta\mathfrak{f}\mathfrak{b}}$.

Example 21. The following theorem in $\mathfrak{M}_{\beta\mathfrak{f}\mathfrak{b}}$ characterizes the fact that in all Henkin models we have exactly four functions mapping truth values to truth values.

$$((p\ \lambda X_o . X_o) \wedge (p\ \lambda X_o . \neg X_o) \wedge (p\ \lambda X_o . \bot) \wedge (p\ \lambda X_o . \top)) \Rightarrow \forall Y_{o\to o} . (p\ Y)$$

Example 22. As exploited in [11], set theory problems can be concisely and elegantly formulated in higher-order logic when using λ-abstraction to encode sets as characteristic functions. For instance, given a predicate $p_{\alpha\to o}$ the set of all objects of type α that have property p is denoted as $\lambda X_\alpha . (pX)$. We then define set operations as follows (we give only some examples):

set operation	defined by
$\in_{\alpha\to(\alpha\to o)\to o}$	$\lambda Z_\alpha \lambda X_{\alpha\to o}(XZ)$
$\{.\}_{\alpha\to(\alpha\to o)}$	$\lambda U_\alpha(\lambda Z_\alpha . Z \stackrel{*}{=} U)$
$\emptyset_{\alpha\to o}$	$(\lambda Z_\alpha \bot)$
$\cap_{(\alpha\to o)\to(\alpha\to o)\to(\alpha\to o)}$	$\lambda X_{\alpha\to o}\lambda Y_{\alpha\to o}(\lambda Z_\alpha . Z \in X \wedge Y \in Y)$
$\cup_{(\alpha\to o)\to(\alpha\to o)\to(\alpha\to o)}$	$\lambda X_{\alpha\to o}\lambda Y_{\alpha\to o}(\lambda Z_\alpha . Z \in X \vee Y \in Y)$
$\subseteq_{(\alpha\to o)\to(\alpha\to o)\to o}$	$\lambda X_{\alpha\to o}\lambda Y_{\alpha\to o}(\forall Z_\alpha . Z \in X \Rightarrow Y \in Y)$
$\wp_{(\alpha\to o)\to((\alpha\to o)\to o)}$	$\lambda X_{\alpha\to o}(\lambda Y_{\alpha\to o} . Y \subseteq X)$

We can now formulate some test problems on sets:

(a) $a_{\alpha\to o} \cup (b_{\alpha\to o} \cap c_{\alpha\to o}) \stackrel{set}{=} (a \cup b) \cap (a \cup c)$ is valid in $\mathfrak{M}_\beta$.
(b) $a_{\alpha\to o} \cup (b_{\alpha\to o} \cap c_{\alpha\to o}) \stackrel{*}{=} (a \cup b) \cap (a \cup c)$ is valid in $\mathfrak{M}_{\beta\xi\mathfrak{b}}$ but not in model classes without ξ and $\mathfrak{b}$.
(c) $\wp(\emptyset_{\alpha\to o}) \stackrel{set}{=} \{\emptyset_{\alpha\to o}\}$ is valid in $\mathfrak{M}_{\beta\mathfrak{f}\mathfrak{b}}$ but not in model classes without $\mathfrak{f}$ and $\mathfrak{b}$. The example is not valid in $\mathfrak{M}_\beta$ due to the embedded equation introduced by the definition of a singleton set $\{.\}$.
(d) and $\wp(\emptyset_{\alpha\to o}) \stackrel{*}{=} \{\emptyset_{\alpha\to o}\}$ is valid in $\mathfrak{M}_{\beta\mathfrak{f}\mathfrak{b}}$ but not in model classes without $\mathfrak{f}$ and $\mathfrak{b}$.

These examples motivate pre-processing in higher-order theorem proving in which the definitions are fully expanded and in which the extensionality principles are employed es early as possible. After pre-processing, many problems of this kind can be automatically translated from their concise and human readable higher-order representation into first-order or even propositional logic representations to be easily checked by respective specialist systems.

5.3 Set Comprehension

One of the advantages of Church's type theory is that instead of assuming comprehension axioms one can simply use terms defining sets for set instantiations. Such set instantiations make use of logical constants in the signature Σ. As in [14] one can vary the signature of logical constants in order to vary the set comprehension assumed in Σ-models. With different amounts of set comprehension, different examples will be valid.

Generating set instantiations is one of the toughest challenges for the automation of higher-order logic. (In fact set instantiations can be employed to simulate the cut-rule as soon as one of the following prominent axioms of higher-order logic is available in the search space: comprehension, induction, extensionality, choice, description.) Set instantiations are often generated during automated search using an enumeration technique involving primitive substitutions.

For each example below, we note restrictions on the signature Σ under which the example is either valid or not valid. Since we would like to distinguish between signatures which contain primitive equality (at various types) and those which do not, we consider classes of models which do not necessarily satisfy property $\mathfrak{q}$. In particular, let $\mathfrak{M}_\beta^{-\mathfrak{q}}(\Sigma)$ be the set of all Σ-models and let $\mathfrak{M}_{\beta\mathfrak{f}\mathfrak{b}}^{-\mathfrak{q}}(\Sigma)$ be the set of all Σ-models satisfying properties $\mathfrak{f}$ and $\mathfrak{b}$ (without requiring property $\mathfrak{q}$).

As in Example 8 one can focus on the use of logical constants in Σ for instantiations and ignore certain uses of logical constants to express the formula. For example, suppose $\mathbf{A} \in \mathit{cwff}_o(\Sigma)$, $\mathcal{M}$ is a Σ-model and $\neg \notin \Sigma$. While $(\neg\mathbf{A}) \notin \mathit{wff}_o(\Sigma)$, we can consider $(\neg\mathbf{A})$ to be a *Σ-external proposition* and define $\mathcal{M} \models \neg\mathbf{A}$ to mean $\mathcal{M} \not\models \mathbf{A}$. Intuitively, the negation is used *externally* in $(\neg\mathbf{A})$. We can inductively define the set of Σ-external propositions $\mathbf{M}$ and the meaning of $\mathcal{M} \models \mathbf{M}$ for Σ-models $\mathcal{M}$. After doing so, most of the examples below are Σ-external propositions even if Σ contains no logical constants. Only Examples 30 and 33 in this section make nontrivial uses of certain logical constants to express the propositions. Due to space considerations, we refer the reader to [14] for details.

If Σ is sufficiently small, then one can construct two trivial models in $\mathfrak{M}^{-\mathfrak{q}}_{\beta\mathfrak{fb}}(\Sigma)$ where $\mathcal{D}_o$ is either simply $\{\mathtt{T}\}$ or $\{\mathtt{F}\}$. (This possibility was ruled out in [8] since we assumed $\neg \in \Sigma$.)

Example 23. $\exists PP$ is valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\Sigma)$ if either $\top \in \Sigma$ or $\neg \in \Sigma$. The example is not valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta\mathfrak{fb}}(\Sigma)$ if $\Sigma \subseteq \{\bot, \wedge, \vee\} \cup \{\Pi^\alpha, \Sigma^\alpha | \alpha \in \mathcal{T}\}$. (Any proof must use a set instantiation involving either $\top$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ or some primitive equality.)

Example 24. $\neg\forall PP$ is valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\Sigma)$ if either $\bot \in \Sigma$ or $\neg \in \Sigma$. The example is not valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta\mathfrak{fb}}(\Sigma)$ if $\Sigma \subseteq (\overline{\Sigma} \setminus \{\bot, \neg\})$. (Any proof must use a set instantiation involving either $\bot$ or $\neg$.)

Example 25 characterizes when an instantiation satisfying the property of negation is possible. This can be either because the signature supplies negation or supplies enough constants to define negation.

Example 25. $\exists N_{o\to o}\forall P_o.NP \Leftrightarrow \neg P$ is valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\Sigma)$ if $\neg \in \Sigma$. The example is also valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\Sigma)$ if $\bot \in \Sigma$ and $\{\Rightarrow, \Leftrightarrow\} \cap \Sigma \neq \emptyset$ since one can consider either the term $\lambda X_o.X \Rightarrow \bot$ or the term $\lambda X_o.X \Leftrightarrow \bot$. The example is not valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta\mathfrak{fb}}(\Sigma)$ if $\Sigma \subseteq \{\top, \bot, \wedge, \vee\} \cup \{\Pi^\alpha, \Sigma^\alpha | \alpha \in \mathcal{T}\}$.

One possibility we did not cover in Example 25 is if Σ is $\{\bot, =^o\}$. Consider the term $(\lambda X_o.X =^o \bot)$. This only defines negation if we assume Boolean extensionality. Hence we obtain the interesting fact that Example 25 is valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta\mathfrak{fb}}(\{\bot, =^o\})$, but is *not* valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\{\bot, =^o\})$.

One can modify Example 25 in a way that requires not only a set instantiation for negation, but also extensionality.

Example 26. $\neg\forall F_{o\to o}\exists X.(FX) \stackrel{*}{=} X$ is valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta\mathfrak{fb}}(\Sigma)$ if $\neg \in \Sigma$. The example is not valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\Sigma)$ regardless of the signature Σ. Also, the example is not valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta\mathfrak{fb}}(\Sigma)$ if $\Sigma \subseteq \{\top, \bot, \wedge, \vee\} \cup \{\Pi^\alpha, \Sigma^\alpha | \alpha \in \mathcal{T}\}$.

Example 27 characterizes when an instantiation can essentially define disjunction and Example 28 characterizes when an instantiation can essentially define the universal quantifier at type α. Clearly one can modify these examples for any other logical constant.

Example 27. $\exists D_{o\to o\to o}\forall P_o\forall Q_o.DPQ \Leftrightarrow (P \vee Q)$ is valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\Sigma)$ if $\vee \in \Sigma$. The example is also valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\Sigma)$ if $\{\neg, \wedge\} \subseteq \Sigma$.

Example 28. $\exists Q_{(\alpha\to o)\to o)}\forall P_{\alpha\to o}.QP \Leftrightarrow \forall X_\alpha.PX$ is valid in $\mathfrak{M}^{-\mathfrak{q}}_{\beta}(\Sigma)$ if $\Pi^\alpha \in \Sigma$.

Recall that Example 8 already provided an example in which one *might* require a set instantiation involving primitive equality (depending on how the calculus relates Leibniz equality to primitive equality).

A few interesting set instantiations involve no logical constants, but do make use of projections (see [18]). Sometimes such projections can be obtained from higher-order unification, as in Example 29.

Example 29. $\exists N_{o\to o}\forall P_o.NP \Leftrightarrow P$ is valid in $\mathfrak{M}_\beta^{-\mathfrak{q}}(\emptyset)$.

However, one cannot expect higher-order unification to *always* provide projection terms when they are needed. Example 30 was studied extensively in [2] (see THM104) in order to demonstrate this fact. In this example, we make use of the abbreviation $\{.\}$ which was defined in Example 22. If the definition of $\{.\}$ makes use of primitive equality, one must assume $=^\iota \in \Sigma$ to express the proposition. If $\{.\}$ is defined using Leibniz equality, then one must assume $\neg, \Pi^{\iota\to o} \in \Sigma$ to express the proposition.

Example 30. $\forall X_\iota \forall Z_\iota.\{X\}\doteq\{Z\} \Rightarrow X\doteq Z$ is valid in $\mathfrak{M}_\beta^{-\mathfrak{q}}(\Sigma)$ so long as Σ is sufficient to express the proposition.

The examples above are straightforward examples designed to ensure completeness of theorem provers with respect to set comprehension. A more natural theorem which requires set instantiations is Cantor's Theorem. Two forms of Cantor's Theorem were studied with respect to set comprehension in [14]. Example 31 is the surjective form of Cantor's Theorem discussed in [4].

Example 31. (Surjective Cantor Theorem) $\neg\exists G_{\alpha\to\alpha\to o}\forall F_{\alpha\to o}\exists J_\alpha.GJ =^{\alpha\to o} F$ is valid in $\mathfrak{M}_{\beta\mathfrak{fb}}^{-\mathfrak{q}}(\Sigma)$ if $\neg \in \Sigma$. The example is not valid in $\mathfrak{M}_{\beta\mathfrak{fb}}^{-\mathfrak{q}}(\Sigma)$ if $\Sigma \subseteq \{\top, \bot, \wedge, \vee\} \cup \{\Pi^\alpha, \Sigma^\alpha | \alpha \in \mathcal{T}\}$ (see Theorem 6.7.8 in [14]).

An alternative formulation of Cantor's Theorem (see [5,14]) is the injective form shown in Example 32. Almost any higher-order theorem prover complete for the corresponding model class should be capable of proving the previous examples in this subsection. Example 32 is far more challenging. At the present time, no theorem prover has found a proof of Example 32 automatically.

Example 32. (Injective Cantor Theorem) $\neg\exists H_{(\iota\to o)\to\iota}\forall P_{\iota\to o}\forall Q_{\iota\to o}.HP =^\iota HQ \Rightarrow P =^{\iota\to o} Q$ is valid in $\mathfrak{M}_{\beta\mathfrak{fb}}^{-\mathfrak{q}}(\Sigma)$ if $\{\neg, \wedge, =^\iota, \Pi^{\iota\to o}\} \subseteq \Sigma$ (see Lemma 6.7.2 in [14]). The example is not valid in $\mathfrak{M}_{\beta\mathfrak{fb}}^{-\mathfrak{q}}(\Sigma)$ if $\Sigma \subseteq \{\top, \bot, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \Pi^\iota, \Sigma^\iota, =^{\iota\to o}\}$. (This fact follows from the results in Section 6.7 of [14].)

One of the difficulties of proving Example 32 is that certain set instantiations seem to be needed beneath other set instantiations (see [5]). The next family of examples illustrates that nontrivial set instantiations can occur within set instantiations with an arbitrary number of iterations.

Example 33. Assume Σ contains $\neg$ and Π^α for every type α. Fix a constant c_ι. We will define a theorem $\mathbf{D}_o^n$ for each natural number n. By induction on n, define simple types τ^n and abbreviations $\mathbf{A}^n_{\tau^n\to o}$ as follows.

(a) Let τ^0 be the type ι and τ^{n+1} be $\tau^n \to o$ for each natural number n.
(b) Let $\mathbf{A}^0_{\iota\to o}$ be $\lambda Z_\iota.(Z\doteq c_\iota) \wedge \top$ and $\mathbf{A}^{n+1}$ be $\lambda Z_{\tau^{n+1}}.(Z\doteq\mathbf{A}^n) \wedge \exists T_{\tau^n}.ZT$ for each natural number n.

Finally, for each n, let $\mathbf{D}_o^n$ be $\exists S_{\tau^n} A^n S$. Each $\mathbf{D}^n$ is a valid in $\mathfrak{M}_\beta^{-\mathfrak{q}}(\Sigma)$. The constant c_ι is the obvious witness for $\mathbf{D}^0$. For each n, $\mathbf{A}^n$ is the witness for $\mathbf{D}^{n+1}$. Note that a subgoal of showing $\mathbf{A}^n$ is the witness for $\mathbf{D}^{n+1}$ involves showing $\mathbf{A}^n$ is nonempty (which was $\mathbf{D}^n$). Hence this proof of $\mathbf{D}^{n+1}$ involves all the previous instantiations $\mathbf{A}^0, \ldots, \mathbf{A}^n$.

6 More Complex Examples

Here we present technically or proof theoretically challenging examples. First we consider a class of hard problems simply involving β-reduction.

Example 34. Let α^0 be ι and α^{n+1} be $(\alpha^n \to \alpha^n)$ for each n. Note that the Church numeral $\overline{2}^{\alpha^n}$ has type α^{n+2}. For any n we can form the term $(\overline{2}^{\alpha^n}\overline{2}^{\alpha^{n-1}}\cdots\overline{2}^{\alpha^0})$ of type $(\iota \to \iota) \to \iota \to \iota$. The size of the β-normal form of this term is approximately of size $2^{(2^{\cdots}2)}$ containing $n+1$ '2s'. (This is a well-known example, mentioned in [27].) For $n \geq 4$ it becomes infeasible to β-normalize such a term (since $2^{2^{2^2}}$ is 2^{65536}, a number much larger than google). One can express relatively simple theorems using this term such as

$$(\overline{2}^{\alpha^n}\overline{2}^{\alpha^{n-1}}\cdots\overline{2}^{\alpha^0})(\lambda X_\iota X) \stackrel{*}{=} (\lambda X_\iota X).$$

If one avoids eager β-normalization and allows lemmas, then there is a reasonably short proof using higher-order logic. We first define the set C_2^α of Church numerals (over α) greater than or equal to 2:

$$\lambda N_{(\alpha\to\alpha)\to\alpha\to\alpha}\forall P.(P\overline{2}^\alpha \wedge (\forall M.PM \Rightarrow P(\overline{s}M))) \Rightarrow PN.$$

(Technically, $(\overline{0}\ \overline{2})$ is β-equal to $(\lambda F_{\iota\to\iota}F)$, which is not equal to $\overline{1}$. We work with the set of Church numerals greater than or equal to 2 to avoid this problem.) One can prove two results with little trouble (where the lengths of the proofs do not depend on the type α):

(a) $\forall N_{((\alpha\to\alpha)\to\alpha\to\alpha)\to(\alpha\to\alpha)\to\alpha\to\alpha}.C_2^{\alpha\to\alpha}N \Rightarrow C_2^\alpha(N\overline{2}^\alpha)$
(b) $\forall N_{(\alpha\to\alpha)\to\alpha\to\alpha}.C_2^\alpha N \Rightarrow .(N(\lambda X_\alpha X)) = (\lambda X_\alpha X)$

Using (a) at several types and (b) at type ι, we can prove, e.g.,

$$(\overline{2}^{\alpha^4}\overline{2}^{\alpha^3}\overline{2}^{\alpha^2}\overline{2}^{\alpha^1}\overline{2}^{\alpha^0})(\lambda X_\iota X) \stackrel{*}{=} (\lambda X_\iota X)$$

in higher-order logic without β-normalizing.

In [13, Chapter 25, p. 376–382] Boolos presents a related example of a first-order problem which has only a very long (practically infeasible) derivation in first-order logic, but which has a short derivation in a second-order logic, by making use of comprehension axioms.

Example 35. (Boolos' Curious Inference)

$$\begin{aligned}&(\forall n.f(n,1) = s(1) \wedge \forall x.f(1,s(x)) = s(s(f(1,x)))\\ &\quad\wedge \forall n.\forall x.f(s(n),s(x)) = f(n,f(s(n),x))\\ &\quad\wedge D(1) \wedge \forall x.(D(x) \Rightarrow D(s(x))))\\ &\Rightarrow D(f(s(s(s(1)))),s(s(s(s(1))))))\end{aligned}$$

If there were an appropriate (first-order) induction principle available, then there should be a short proof of this example. Note that the example specifies f to be the Ackermann function which grows extremely fast and hence $f(s(s(s(1)))),s(s(s(1)))))$

is a very big number. Actually, there is long first-order proof which is relatively easy to describe. Boolos argues that any first-order proof must be of size at least $2^{(2^{\cdots 2})}$ containing 64K '2s' in all (far more enormous than the number 2^{64K} in Example 34). There is no chance of formally representing such a proof with all computation power ever. Boolos presents a short alternative proof in second-order logic that makes use of higher-order lemmas obtained from comprehension axioms. Formulating the appropriate lemmas (as with the lemmas in Example 34) requires human ingenuity that goes beyond the capabilities of what can be supported with primitive substitution and lemma speculation techniques in current theorem proving approaches.

As discussed in [3], there is a family of theorems $\mathbf{A}^1, \mathbf{A}^2, \ldots$ which are all of the same low order such that $\mathbf{A}^n$ is not provable unless one uses set instantiations involving n^{th}-order quantifiers. To obtain concrete examples from the argument, one must use Gödel numbering. A family of simpler examples displaying this phenomenon would likely be enlightening.

7 Conclusion

We have presented a first set of higher-order test examples that may support the development of higher-order proof systems. This set of examples has been structured according to technical aspects and the semantic properties of extensionality and set comprehension. Future work is to add examples and include them in either the TPTP library or an appropriate higher-order variant. Many more examples are particularly needed to illustrate properties of different forms of equality.

References

1. P. B. Andrews. General models and extensionality. *J. of Symbolic Logic*, 37(2):395–397, 1972.
2. P. B. Andrews. On Connections and Higher Order Logic. *J. of Automated Reasoning*, 5:257–291, 1989.
3. P. B. Andrews. Classical type theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 15, pages 965–1007. Elsevier Science, 2001.
4. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, second edition, 2002.
5. P. B. Andrews, M. Bishop, and C. E. Brown. TPS: A theorem proving system for type theory. In D. McAllester, editor, *Proc. of CADE-17*, number 1831 in LNAI, pages 164–169, Pittsburgh, USA, 2000. Springer.
6. Peter B. Andrews. Resolution in type theory. *J. of Symbolic Logic*, 36(3):414–432, 1971.
7. C. Benzmüller. *Equality and Extensionality in Automated Higher-Order Theorem Proving*. PhD thesis, Saarland University, 1999.
8. C. Benzmüller, C. Brown, and M. Kohlhase. Higher-order semantics and extensionality. *J. of Symbolic Logic*, 69(4):1027–1088, 2004.
9. C. Benzmüller, C. E. Brown, and M. Kohlhase. Semantic techniques for higher-order cut-elimination. SEKI Technical Report SR-2004-07, Saarland University, Saarbrücken, Germany, 2004. Available at: `http://www.ags.uni-sb.de/~chris/papers/R37.pdf`.

10. C. Benzmüller and M. Kohlhase. LEO – a higher order theorem prover. In C. Kirchner and H. Kirchner, editors, *Proc. of CADE-15*, number 1421 in LNAI, pages 139–144, Lindau, Germany, 1998. Springer.
11. C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Can a higher-order and a first-order theorem prover cooperate? In F. Baader and A. Voronkov, editors, *Proc. of LPAR 2004*, volume 3452 of *LNAI*, pages 415–431. Springer, 2005.
12. M. P. Bonacina and J. Hsiang. Incompleteness of the RUE/NRF inference systems. Newsletter of the Association for Automated Reasoning, No. 20, pages 9–12, 1992.
13. G. Boolos. *Logic, Logic, Logic*. Harvard University Press, 1998.
14. C. E. Brown. *Set Comprehension in Church's Type Theory*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 2004.
15. A. Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.
16. V. J. Digricoli. Resolution by unification and equality. In W. H. Joyner, editor, *Proc. of CADE-4*, Austin, Texas, USA, 1979.
17. Leon Henkin. Completeness in the theory of types. *J. of Symbolic Logic*, 15(2):81–91, 1950.
18. G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
19. M. Kohlhase. *A Mechanization of Sorted Higher-Order Logic Based on the Resolution Principle*. PhD thesis, Saarland University, 1994.
20. M. Kohlhase. Higher-order tableaux. In *Proc. of TABLEAUX 95*, number 918 in LNAI, pages 294–309. Springer, 1995.
21. J.D. McCharen, R.A. Overbeek, and L.A. Wos. Problems and Experiments for and with Automated Theorem-Proving Programs. *IEEE Transactions on Computers*, C-25(8):773–782, 1976.
22. D. Miller. *Proofs in Higher-Order Logic*. PhD thesis, Carnegie-Mellon Univ., 1983.
23. F.J. Pelletier. Seventy-five Problems for Testing Automatic Theorem Provers. *J. of Automated Reasoning*, 2(2):191–216, 1986.
24. F.J. Pelletier, G. Sutcliffe, and C.B. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.
25. C. Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. Progress in Theoretical Computer Science. Birkhäuser, 1998.
26. W. Snyder and J. Gallier. Higher-Order Unification Revisited: Complete Sets of Transformations. *J. of Symbolic Computation*, 8:101–140, 1989.
27. R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.
28. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *J. of Automated Reasoning*, 21(2):177–203, 1998.
29. G.A. Wilson and J. Minker. Resolution, Refinements, and Search Strategies: A Comparative Study. *IEEE Transactions on Computers*, C-25(8):782–801, 1976.
30. C.-P. Wirth. Descente infinie + Deduction. *Logic J. of the IGPL*, 12(1):1–96, 2004. `www.ags.uni-sb.de/~cp/p/d/welcome.html`.

Formal Modeling of a Slicing Algorithm for Java Event Spaces in PVS

Néstor Cataño

Department of Computer Science, The University of York, U.K
catano@cs.york.ac.uk

Abstract. This paper presents the formalization of an algorithm for slicing Java event spaces in PVS. In short, Java event spaces describe how multi-threaded Java programs operate in memory. We show that Java event spaces can be sliced following an algorithm introduced in previous work and still preserve properties in a subset of CTL. The formalization and proof presented in this paper can be extended to other state-space reduction techniques as long as some *sufficient* conditions are fulfilled.

1 Introduction

Java event spaces [2,10] are partial orders of the actions performed by the main memory and the threads of a multi-threaded Java program. In previous work [1] we showed how classical slicing techniques can be employed to reduce the size of Java event spaces. Roughly speaking, when slicing, only the parts of the Java event space upon which the elements of the slicing criterion depend are retained, while the underlying structure of the Java event space is preserved. Furthermore, we dealt with the problem of *aliasing* that arises when two variables of the event space point to the same memory address. An algorithm that takes a Java event space and calculates aliasing dependencies for relevant variables of the slicing criterion was outlined.

Here, we formalize the aliasing algorithm in PVS [8] and, for parts of the algorithm, show how Java event spaces can be sliced and still verify the same properties in CTL without the *next* operator [4,5] as non-sliced Java event spaces. To cope with the proof, we propose a two-step trace reconstruction approach. This process of reconstruction outlines conditions which must be verified for other algorithms working with state-space reduction to preserve properties in CTL. The outline of these conditions and the PVS formalization are the two main contributions of this paper.

This paper is structured as follows. Section 2 introduces Java event spaces formally and gives an example where an event space for a multi-threaded Java program is calculated. Section 3 presents the slicing algorithm introduced in [1]. Section 4 formalizes Java event spaces as finite-state automata. This allows for defining how CTL properties are evaluated on Java event spaces. Section 5 shows that the algorithm introduced in [1] preserves properties expressed in a subset

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 82–97, 2005.

of CTL. During the proof we identify sufficient conditions employable in similar proofs when different kinds of state-space algorithms are applied. Finally, Section 6 gives conclusions and presents future work.

2 Java Event Spaces

Chapter 17 of the Java Language Specification (JLS) [7] gives a detailed yet not formal specification of how multi-threaded Java programs should operate. This specification states that a main memory shared by all the threads in the program exists, and that it keeps a global copy of the variable values. The specification also says that each thread has its own local working memory which keeps a copy of variables of the main memory. As a thread executes code, some *events* in memory happen. An event in memory represents the occurrence of some *action* either in the main memory or in the working memory of some thread. A thread θ can for example use the *right value* v of a *left value* l, action $\mathbf{use}(\theta, l, v)$, or it can assign it a new value v, action $\mathbf{assign}(\theta, l, v)$. Right values represent object values as seen in memory: native type values and references. Left values are memory addresses. When copying the value v of l from the main memory to the working memory of θ, two actions must occur: first, a $\mathbf{read}(\theta, l, v)$ action performed by the main memory, followed at some unspecified time later by a $\mathbf{load}(\theta, l, v)$ action performed by the working memory. When copying the value v of l from the working memory of θ to the main memory, two actions must occur as well: a $\mathbf{store}(\theta, l, v)$ action performed by the working memory, followed at some unspecified time later by a $\mathbf{write}(\theta, l, v)$ action performed by the main memory. Actions $\mathbf{lock}(\theta, o)$ and $\mathbf{unlock}(\theta, o)$ acquire and relinquish a lock on the object o on behalf of the thread θ.

We use the notation $x : \mathbf{y}$ to indicate that x is an event labeled with an action $\mathbf{y}$. Memory actions are **read**, **write**, **lock** and **unlock**; thread actions are **load**, **use**, **assign**, **store**, **write**, **lock** and **unlock**; and lock actions **lock** and **unlock**. With $x{:}\,\mathbf{read}(l)$ we indicate that $x{:}\,\mathbf{read}(\theta, l, v)$ for some θ and v. Analogously $x{:}\,\mathbf{write}(v)$ means that $x{:}\,\mathbf{write}(\theta, l, v)$ for some θ and l. Similarly for the other actions. We use $ref_{\mathtt{x}}$ to indicate the reference associated with variable `x`.

Formally expressed, a Java event space is a set of events X labeled with actions, provided with a partial order $\leq$, such that $(X, \leq)$ respects the rules of well-formedness enunciated in the specification of the Java Memory Model (JMM) [2,10]. We give an example of event space generation for a Java program that describes the interaction of two threads executing two methods in parallel.

Example 1. Suppose that two threads θ_1 and θ_2, executing respectively methods `p()` and `q()` on some object `this`, exist.

```
void p(){ synchronized(this){x.i = 7; x.j = 5;} y.i = x.j; }
void q(){ synchronized(this){y = x; z.i = y.i;} z.i = 9; }
```

Further, suppose that variables `x`, `y` and `z` are instances of some class `C` with variables `i` and `j` of type `int`, whose initial values are 0 for both. Figure 1

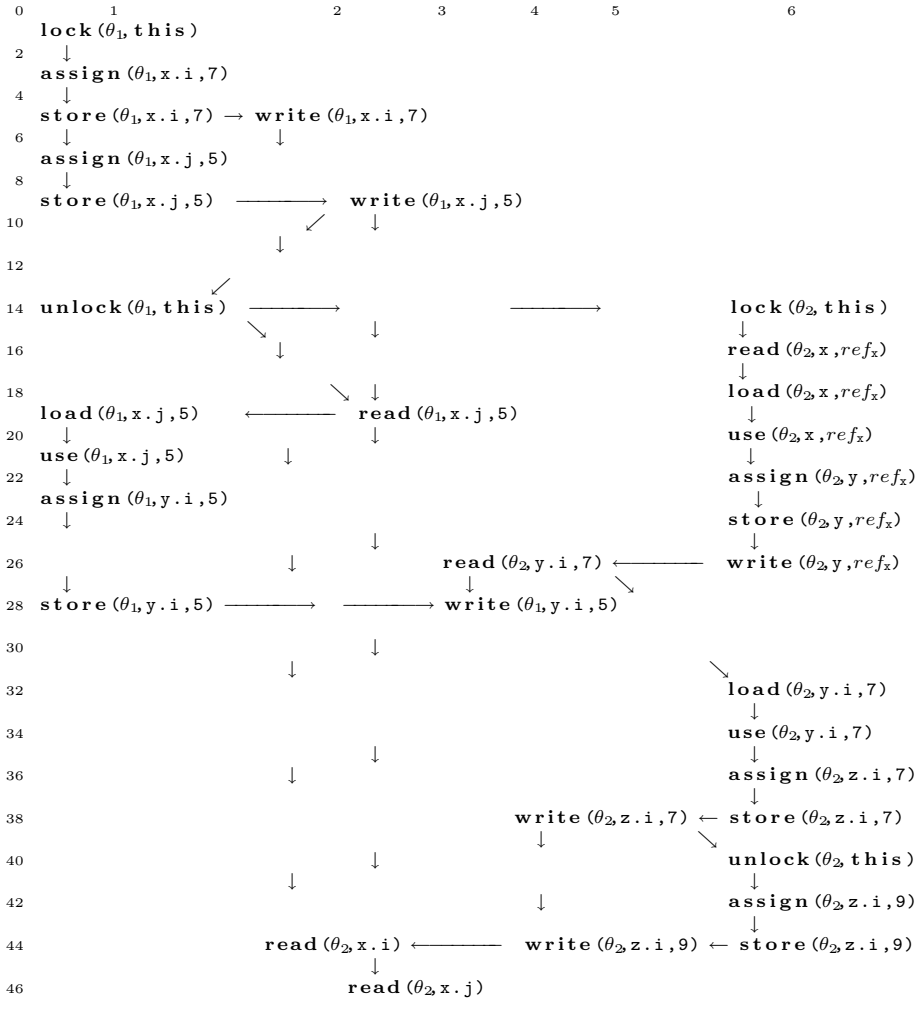


Fig. 1. Java event space generation

shows an event space for the interaction of actions generated by both the main memory and the working memories local to θ_1 and θ_2. This event space represents a possible execution of the program for the interaction of these two threads. The partial order relation $\leq$ of actions is represented in the figure by (multiple) vertical, horizontal and diagonal arrows.

For readability, the reflexivity and transitivity of the partial order relation have not been sketched. Because, according to the JMM, thread actions for the same thread make up a total order, actions for θ_1 and θ_2 in Columns 1 and 6 form increasing chains. The same thing happens for memory actions on the same variable, hence actions **read** and **write** for variables `x.i`, `x.j`, `y.i` and `z.i` in Columns 2 to 5 also form increasing chains.

In Java, when a thread is executing a synchronized code fragment of some object, no other thread may acquire a lock associated with the same object. Though, a single thread may acquire several times the lock associated with the same object. In our example, `p()` and `q()` are synchronized. We sketch the case when θ_1 acquires the lock on object `this` before θ_2. After acquiring the lock associated with `this` (Column 1, Line 1), θ_1 executes the body of the synchronized part of `p()` (Lines 3 to 9), and finally relinquishes the lock (Column 1, Line 14). In Line 3, θ_1 assigns `7` to `x.i` and, in Line 5, it writes the new value of `x.i` from its working memory to the main memory. The thread θ_1 then executes the rest of the synchronized part of `p()`, since θ_2 cannot acquire a lock on the object `this` in order to execute the synchronized part of `q()`.

From then on θ_1 and θ_2 continue to execute concurrently, in particular the Java Language Specification does not specify whether the locking of `this` on behalf of θ_2 or the execution of `y.i = x.j` by θ_1 occurs first. One can only be sure that *writing to* and *reading from* a certain variable must respect a total order. Figure 1 sketches the case when reading from `y.i` in the synchronized statement of `q()` occurs before the writing to `y.i` in `p()` (Column 4). In Column 6, θ_2 reads the value of `x` from the main memory (Lines 16 and 18), uses its value (Line 20), and finally assigns the reference of `x` to `y` (Line 22). Consequently, from there on, `x` and `y` will be *aliased*. The rest of Column 6 shows the memory interactions corresponding to the execution of `z.i = 9;` by θ_2.

3 The Slicing Algorithm

A *program slice* consists of those parts of a program that potentially affect some points of interest, which in turn depend on the property that is checked. These points of interest are called *slicing criterion* and its variables *relevant variables*. A *slice set* S_C is composed of nodes in the Java event space from which nodes in the slicing criterion C are reachable via the dependency relation $\xrightarrow{fd_a}$, defined below, with the intuitive meaning $w \xrightarrow{fd_a} r$ if the event r is *aliasing flow dependent* on the event w. In addition to elements in S_C, a *residual slice set* S_{Cr} must contain other events, so that the Java event space formed of events in S_{Cr} and having as order relation the partial order relation of the event space restricted to S_{Cr} make up a *program slice*.

Predicate *FlowDep*? below constitutes the basis for the slicing algorithm introduced in [1]. This predicate formalizes the notion of aliasing flow dependency $\xrightarrow{fd_a}$; more concretely $w \xrightarrow{fd_a} r$ is given by $FlowDep?(w, r)$. In the first case of definition of *FlowDep*?, when `y=x` and w "occurs before" r ($w \leq r$), **read**(`x.i`) is alias flow dependent of **write**(`y.i`) if there is no **write** action w_1 (other than w) between w and r that modifies the field `i` of `x`. When `y` and `x` are distinct two other cases arise. First, if $w \leq r$, it is not only necessary to ensure that there is no **write** action w_1 between w and r modifying `x.i`, but also that `y` is an alias of x when w happened. When `y` and `x` are different, it is possible that w:**write**(`y.i`) and r:**read**(`x.i`) are not related, since the e Java Language Specification does not

ensure a total order for **write** and **read** actions on different left values. In this case a *defensive* approach is adopted by considering that **write**(y.i) modifies x.i provided that y is an alias of x when w happened.

$$FlowDep?(w{:}\,\mathbf{write}(\mathtt{y.i}), r{:}\,\mathbf{read}(\mathtt{x.i})) = \begin{cases} true, \text{ if } \begin{cases} 1.\ \mathtt{y}{=}\mathtt{x} \ \wedge \ w{\leq}r \ \wedge \\ 2.\ \neg\exists w_1{:}\,\mathbf{write}(\mathtt{z.i}).Alias?(\mathtt{x},\mathtt{z},w_1) \wedge w{<}w_1{\leq}r \wedge FlowDep?(w_1,r) \end{cases} \\ true, \text{ if } \begin{cases} 1.\ \mathtt{y}{\neq}\mathtt{x} \ \wedge \ w{\leq}r \ \wedge \\ 2.\ Alias?(\mathtt{x},\mathtt{y},w) \ \wedge \\ 3.\ \neg\exists w_1{:}\,\mathbf{write}(\mathtt{z.i}).Alias?(\mathtt{x},\mathtt{z},w_1) \wedge w{<}w_1{\leq}r \wedge FlowDep?(w_1,r) \end{cases} \\ true, \text{ if } \begin{cases} 1.\ \mathtt{y}{\neq}\mathtt{x} \ \wedge \ w{\not\leq}r \ \wedge \ r{\not\leq}w \ \wedge \\ 2.\ Alias?(\mathtt{x},\mathtt{y},w) \end{cases} \\ false \quad \text{otherwise} \end{cases}$$

The predicate $FlowDep?$ uses the predicate $Alias?(x, y, w)$ to decide whether x and y are aliased at the moment the action w occurs in the Java event space. This last predicate is defined as the disjunction of the predicate $AliasAux?$ with the parameters swapped. The predicate $AliasAux?(\mathtt{y},\mathtt{x},w)$ checks whether, at the moment w occurs, y references the same address as x, as a consequence of an assignment to y from an alias of x. $AliasAux?(\mathtt{y},\mathtt{x},w)$ holds if ($i.$) y is written to by x — $w_2{:}\,\mathbf{write}(\mathtt{y}, ref_{\mathtt{x}})$ — and the reference of y is not modified afterward by any w_1: **write** to some reference $ref_{\mathtt{t}}$ which is not alias of x, or ($ii.$) y is written to by a z other than x, and z was an alias of x before w_2 occurred.

$$Alias?(\mathtt{x},\mathtt{y},w) = AliasAux?(\mathtt{x},\mathtt{y},w) \ \vee \ AliasAux?(\mathtt{y},\mathtt{x},w)$$

$$\begin{aligned} &AliasAux?(\mathtt{y},\mathtt{x},w{:}\,\mathbf{write}) = \\ &(\exists w_2{:}\,\mathbf{write}(\mathtt{y}, ref_{\mathtt{x}}).w_2{\leq}w \wedge \neg\exists w_1{:}\,\mathbf{write}(\mathtt{y}, ref_{\mathtt{t}}).w_2{\leq}w_1{<}w \wedge \neg Alias?(\mathtt{x},\mathtt{t},w_1)) \vee \\ &(\exists w_2{:}\,\mathbf{write}(\mathtt{y}, ref_{\mathtt{z}}).w_2{<}w \wedge \mathtt{z}{\neq}\mathtt{x} \wedge \\ &\qquad \neg\exists w_1{:}\,\mathbf{write}(\mathtt{y}, ref_{\mathtt{t}}).w_2{<}w_1{<}w \wedge \neg Alias?(\mathtt{x},\mathtt{t},w_1) \wedge Alias?(\mathtt{y},\mathtt{z},w_2)) \end{aligned}$$

Slice sets are formalized by S_C below, where an event w labeled with an action is considered to be in the *carrier* of a Java event space η, $w \in carrier(\eta)$, if the event is related to itself. When S_C is applied to η and an $r{:}\,\mathbf{read}(\mathtt{x.i})$, it returns the set of events $w{:}\,\mathbf{write}(\mathtt{y.i})$ in the *carrier* of η such that the predicate $FlowDep?(\mathbf{write}(\mathtt{y.i}), \mathbf{read}(\mathtt{x.i}))$ holds.

$$S_C(\eta, r{:}\,\mathbf{read}(\mathtt{x.i})) = \{\ w{:}\,\mathbf{write}(\mathtt{y.i}) | w \in carrier(\eta) \wedge FlowDep?(w, r)\ \}$$

Example 2. Given the *slicing criterion* $C = \{\ \mathbf{read}(\mathtt{x.i}),\ \mathbf{read}(\mathtt{x.j})\ \}$ and the event space in Figure 1, $S_C(\mathbf{read}(\mathtt{x.i})) = \{\ \mathbf{write}(\theta_1, \mathtt{x.i}, 7), \mathbf{write}(\theta_1, \mathtt{y.i}, 5)\ \}$ and $S_C(\mathbf{read}(\mathtt{x.j})) = \{\ \mathbf{write}(\theta_1, \mathtt{x.j}, 5)\ \}$. Therefore:

$$S_C = \{\ \mathbf{write}(\theta_1, \mathtt{x.i}, 7),\ \mathbf{write}(\theta_1, \mathtt{x.j}, 5),\ \mathbf{write}(\theta_1, \mathtt{y.i}, 5)\ \}$$

Definition 1 below formalizes the construction of residual slices. We are not going in details on this definition here, but want to indicate that Items ($i.$)

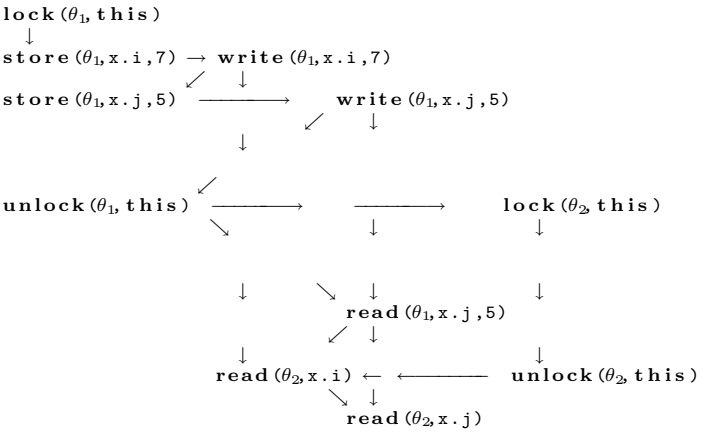


Fig. 2. Sliced event space

through (*iv.*) of this definition rely on the formalization of the JMM [2,10]. For instance, Item (*ii.*) comes from the JMM well-formedness rule *"A thread is not permitted to write data from its working memory back to the main memory for no reason"*. In Definition 1, *storeof* is a function that pairs a **write** action with a unique preceding **store** action.

Definition 1 (residual slice set construction S_{Cr}). Given an event space η and a slice set S_C, the residual set S_{Cr} for S_C is constructed from S_C by adding actions to it as follows:

(*i.*) For each **write** action w: **write**(θ, l, v) in S_C, the only action s in η such that s: **store**(θ, l, v) and $s = storeof(w)$ is added to S_{Cr}.
(*ii.*) For each pair of **store** actions s: **store**(θ, l), s': **store**(θ, l) in η such that $s \neq s'$ and $s \leq s'$, every action a: **assign**(θ, l) in η such that $s \leq a \leq s'$ is added.
(*iii.*) Each **lock** and **unlock** actions in η are added to S_{Cr}.
(*iv.*) For each **write** action w: **write**(l) in S_C, all **read** actions r: **read**(l) in η such that $w \leq r$ or $r \leq w$ are added to S_{Cr}.

Example 3 calculates the residual slice set S_{Cr} for S_C in Example 2.

Example 3. Given S_C as in Example 2, the residual slice set S_{Cr} for the event space in Figure 1 is:

$$S_{Cr} = \{\ \mathbf{write}(\theta_1, \mathtt{x.i}, 7),\ \mathbf{write}(\theta_1, \mathtt{x.j}, 5),\ \mathbf{write}(\theta_1, \mathtt{y.i}, 5), \mathbf{store}(\theta_1, \mathtt{x.i}, 7),$$
$$\mathbf{store}(\theta_1, \mathtt{x.i}, 5),\ \mathbf{store}(\theta_1, \mathtt{y.i}, 5),\ \mathbf{read}(\theta_1, \mathtt{x.j}, 5),\ \mathbf{read}(\theta_2, \mathtt{x.i}),$$
$$\mathbf{read}(\theta_2, \mathtt{y.i}, 7),\ \mathbf{read}(\theta_2, \mathtt{x.j}),\ \mathbf{lock}(\theta_1, \mathtt{this}),\ \mathbf{lock}(\theta_2, \mathtt{this}),$$
$$\mathbf{unlock}(\theta_1, \mathtt{this}), \mathbf{unlock}(\theta_2, \mathtt{this})\ \}$$

Figure 2 presents the event space with events in S_{Cr} preserving the partial order relation in Figure 1.

4 Expressing Java Event Spaces as Finite-State Automata

To check the correctness of a dynamic system, we should be able to specify the kind of properties the system is expected to have. As dynamic systems can be modeled as finite-state transition systems, the formalism behind the specification should be appropriate to express properties about state transitions. *Temporal logic* is a particular formalism suitable to specify properties in terms of sequence of transitions between states in the system. The Computation Tree Logic (CTL) [4,5] is one of the most commonly used temporal logic in model checking. Validity of CTL formulae depends only on the current state of the transition system, this is the reason why CTL formulae are referred to as *state formulae* in literature. CTL formulae are formed of *path quantifiers* and *temporal operators*. Path quantifiers specify that all paths, `A`, or some paths, `E`, starting at some *initial state* have a certain property. Four basic operators exist: `X` (*next*), which requires a property to hold at the second state of the path; `G` (*globally*) requires a property to hold at every state along the path; `F` (*future*) requires a property to hold at some states on the path; and `U` (*until*), combining two properties, which requires that the second property holds at some state along the path and the first holds in any preceding state. CTL requires that each use of a temporal operator be immediately proceeded by the use of a path quantifier. Hence, valid formulae in CTL are in the shape of `EX`ϕ, `EG`ϕ, `EF`ϕ, `E`$[\phi_1$`U`$\phi_2]$, `AX`ϕ, `AG`ϕ, `AF`ϕ, and `A`$[\phi_1$`U`$\phi_2]$.

We are interested in proving that, after the program slice procedure presented in Section 3 is applied to a Java event space, the sliced Java event space verifies the same CTL properties (when the next operator is not considered) as the original Java event space. To prove that, Java event spaces must be formalized as finite-state automata. In the following we present such a formalization in PVS [8]. First Java event spaces are modeled.

Java event spaces. Predicate `IsEventSpace?` below formalizes Java event spaces. A Java event space `E` is an `evtrelation`, *i.e.* a set of pairs of events, that respects the (17) well-formedness rules regarding the JMM enunciated in [7], that is is a partial order — *i.e.* that is reflexive, antisymmetric and transitive — and that has a finite history of elements preceding any event — `FiniteHistory?(E)`. This last predicate holds if for every event `e` in the *carrier* of `E` only a finite number of elements preceding it exists.

```
IsEventSpace?(E:evtrelation) : bool =
  rule1?(E) ∧ ··· ∧ rule17?(E) ∧
  reflexive?(E) ∧ antisymmetric?(E) ∧ transitive?(E) ∧ FiniteHistory?(E)

FiniteHistory?(E:evtrelation) : bool =
 ∀(e:event): is_finite({(d:event)|carrier(E)(e) ∧ carrier(E)(d) ∧ E(d,e)})
```

Java event spaces as finite-state automata. We first define a `store` as an association of right values `rval` to left values `lval`. Then, states are defined as records having two fields: a finite history h of events occurring before reaching

the current state, and a store σ which is updated as events in the history occur. Initial states of the finite-state automaton have an `empty?` history of events and each element of the store has a default value `rdefault`.

```
store: TYPE = [lval → rval]
state: TYPE = [# h: (is_finite[event]), σ: store #]
InitialState: [state → bool] =
  λ(s:state): empty?(s‘h) ∧ s‘σ = λ(l:lval): rdefault
```

Predicate `NextState(E)` below decides whether a one-step transition between two states `s` and `s1` exists; `E` represents the event space to be expressed as a transition system. Formally expressed, `NextState(E)` holds for two states `s` and `s1` if their histories differ by a *single* element `e`, which moreover must belong to the *carrier* of `E`. Additionally, each element `f` in the *carrier* of `E` happening before `e`[1] must be in the history of `s`, and the history and store of `s1` can be obtained respectively from the history and store of `s` when considering only the effect produced by the event `e`. Notice that only `Write` events affect the store. This respects the definition of S_c in Section 3 where only `Write` events are retained in the sliced event space.

```
NextState(E:(IsEventSpace?)): [state,state → bool] =
 λ(s:state,s1:state):
  let {e} = s1‘h \s‘h in
    carrier(E)(e) ∧
    (∀(f:event): carrier(E)(f) ∧ E(f,e) ∧ f/=e ⇒ s‘h(f)) ∧
    s1‘h = s‘h ∪{e} ∧
    s1‘σ = cases e of Write(t,l,r): s‘σ with [l:=r] else s‘σ endcases
```

We can now define whether a trace `tr`, *i.e.* an infinite sequence of states indexed by natural numbers, constitutes a path in a finite-state automaton. First, the initial state of the trace must be an `InitialState`, and a transition between each pair of successive elements `i`, `i+1` of the trace should exist.

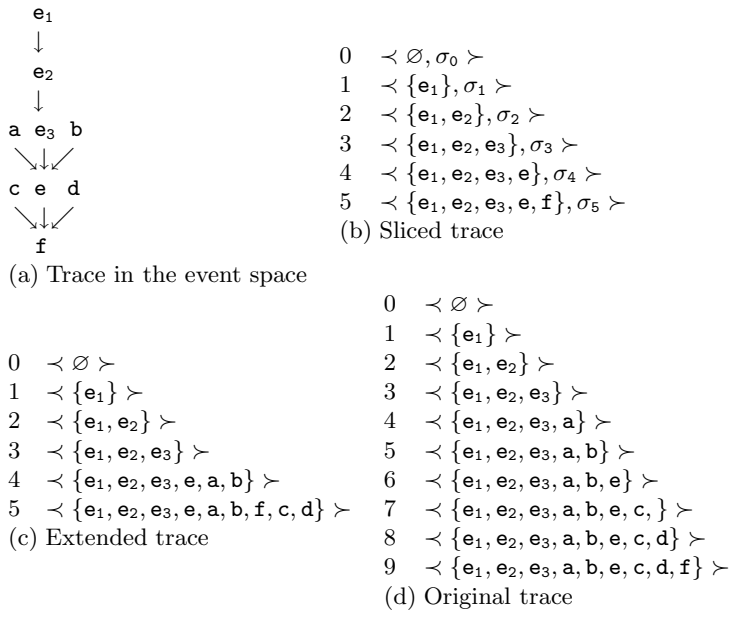
```
trace: TYPE = [nat → state]
Path(E:(IsEventSpace?)): [trace → bool] =
 λ(tr:trace): InitialState(tr(0)) ∧ ∀(i:nat): NextState(E)(tr(i),tr(i+1))
```

Figure 3(b) shows the states transitions of the Java event space in Figure 3(a) after slicing (removing) events `a`, `b`, `c` and `d`. Symbols $\prec\succ$ stand for records of type `state`. Thus, `NextState(E)`$(\prec \{e_1, e_2, e_3\}, \sigma_3 \succ, \prec \{e_1, e_2, e_3, e\}, \sigma_4 \succ)$, for example. Lemmas below follow directly from the definitions of `NextState`, `trace` and `Path`, where `Program_Slice(E,C)` stands for the program slice of the Java event space `E` with respect to the slicing criterion `C`.

The first lemma says that the history for the first state (index `0`) of any sliced trace[2] is empty; the second lemma states that histories of successive states differ

[1] `E(e1,e2)` corresponds to the notation $e_1 \leq e_2$ in the event space `E` used before. Symbol `/=` denotes inequality in PVS, and `s‘h` stands for field h of `s`.

[2] A sliced trace is a trace in a program slice.



(a) Trace in the event space

(b) Sliced trace

(c) Extended trace

(d) Original trace

Fig. 3. Reconstruction of traces in the event space

by a single event; additionally, the third lemma says that these histories grow. The last lemma combines the third and the fourth lemmas.

```
sliced_traces_are_empty_initially: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace):
  Path(Program_Slice(E,C))(tr) ⇒ empty?(tr(0)‘h)

sliced_traces_make_single_steps: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat):
  Path(Program_Slice(E,C))(tr) ⇒
   subset?(tr(i)‘h,tr(i+1)‘h) ∧ singleton?(tr(i+1)‘h \tr(i)‘h)

sliced_traces_are_strict_subsets: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat):
  Path(Program_Slice(E,C))(tr) ⇒ strict_subset?(tr(i)‘h,tr(i+1)‘h)

sliced_traces_as_increments_the : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat):
   Path(Program_Slice(E,C))(tr) ⇒
     tr(i+1)‘h = add(the(tr(i+1)‘h \tr(i)‘h),tr(i)‘h)
```

Evaluating properties. The evaluation of CTL properties follows directly from the standard definitions of CTL operators. For example, given a Java event space E and a state s, property EGϕ holds in s, if a trace tr starting at s — tr(0)=s —

and being a path — `Path(E)(tr)` — exists, such that ϕ is `true` along the trace — $\forall(\mathtt{j} : \mathtt{nat}) : \mathtt{Eval}(\phi)(\mathtt{tr}(\mathtt{j}))$. Further, a property `Holds` provided that it holds at every initial state.

```
Sem(E:(IsEventSpace?)): [property → [state → bool]] =
 λ(prop:property)(s:state):
 cases prop of
  EG(P): ∃(tr:trace): tr(0)=s ∧ Path(E)(tr) ∧ ∀(j:nat): Eval(P)(tr(j)),
   ...
 endcases

Holds(E:(IsEventSpace?)): [property → bool] =
 λ(prop:property): ∀(s:state): InitialState(s) ⇒ Sem(E)(P)(s)
```

5 Program_Slice Is CTL Property-Preserving

We want to prove that, for any proper slicing criterion `C`, if a CTL property `prop` holds in the whole Java event space `E`, then `prop` holds in the sliced Java event space `Program_Slice(E,C)`, and vice-versa. This is expressed in the following two theorems respectively:

```
preserving_slice_fi : theorem
 ∀(E:(IsEventSpace?),C:setof[event],prop:property):
  Holds(E)(prop) ⇒ Holds(Program_Slice(E,C))(prop)

preserving_slice_if : theorem
 ∀(E:(IsEventSpace?),C:setof[event],prop:property):
  Holds(Program_Slice(E,C))(prop) ⇒ Holds(E)(prop)
```

First, notice that slicing can not preserve properties constructed with the aid of the CTL operators `EX` and `AX` because slicing does not preserve *next* states. Second, when using `Holds(R)(prop)` in the definition of `preserving_slice_fi` and `preserving_slice_if` above, a proof of the following lemma, stating that sliced Java event spaces are still Java event spaces, must be first provided.

```
slice_sets_are_event_spaces : lemma
 Program_Slice(E:(IsEventSpace?),C:setof[event]) has_type (IsEventSpace?)
```

This lemma ensures that expressing sliced Java event spaces as finite-state automata according to Section 4 is still valid. We will not focus on the proof of this last lemma here, but will use it in the proof of the second theorem above. This theorem has been proved for the existential operators `EG`, `EU` and `EF`[3]. Our approach considers the reconstruction of traces which incorporate all those events in the event space removed when slicing. For one, this approach allows the making of the whole proof process easier and secondly, if we know that original traces verify the same properties as sliced traces, we are sure that our program

[3] Additionally, the first theorem has been proved for the universal CTL operators `AG`, `AU` and `AF`.

slice is correct in the sense that only those elements that do not change the validity of underlying properties were removed when slicing. This construction is accomplished in two steps.

Constructing original traces from sliced traces. Firstly, we construct *extended traces* from sliced traces `tr`, which modify the history of every state in `tr` in such a way that those events in the event space who relate to any event in the history are added to it (`extended_trace` below gives a precise definition of extended traces). Note that stores on extended traces coincide respectively with stores on sliced traces; that is in accordance with the fact that, when slicing, we rule out only those events that do not affect the store and hence do not affect the validity of the CTL property that is checked. Figure 3(c) shows the extended trace constructed from the sliced trace presented in Figure 3(b). We have intentionally omitted stores as part of states.

```
extended_trace(E:(IsEventSpace?),C:setof[event]):
                                  [(Path(Program_Slice(E,C))) → trace] =
 λ(tr:(Path(Program_Slice(E,C)))) (n:nat):
  (# h:={ e:event | Carrier(E)(e) ∧
                    ∃(a:event): Carrier(E)(a) ∧ tr(n)‘h(a) ∧ E(e,a)},
     σ:= tr(n)‘σ #)
```

We want to make single steps between consecutive indexes, *i.e.* histories between consecutive states should differ by a single element only. However, single steps are not provided by the definition of `extended_trace` (see histories for indexes 3 and 4 in Figure 3(c) for example). To achieve this singleness, events in the history of every state on the extended trace are spelled out, making up *original traces* (see definition below). Figure 3(d) presents the original trace for the extended trace in Figure 3(c). Notice that if the history at index 4 in Figure 3(c) is spelt out, one sole element between `a` or `b` should be chosen from the history first; `e` cannot be chosen because it requires that both `a` and `b` occur before. To make this choice, the *least* between `a` or `b` can be selected; but since, Java event spaces do not provide *total* orders in general, the least between `a` and `b` might not be defined. Assume that such a function `spell_history(E(IsEventSpace?))(k:nat, S:setof[(Carrier(E))]`, spelling the `k` least elements of `S`, exists; as well as `spell_store(E:(IsEventSpace?)) (k:nat,S:setof[event]) (st:strstate)`, which spells the `k` least elements of `S` and return `st` after making it `k` single updates.

From the definition of `original_trace` below, given an index `n`, if `n=0` then `original_trace` returns `tr(0)`. If `n>0`, then `original_trace` takes the minimum index `m` such that `Card(etr(m)‘h)>=n`, where `Card` is the standard cardinality function for sets. If `Card(etr(m)‘h)` is `n`, then the original trace coincides with the extended trace; otherwise, `n-p` — where `p` is `Card(etr(m-1)‘h)` — events are spelt from the difference between `Card(etr(m)‘h)` and `Card(etr(-m-1)‘h)`. The same is done for the store.

```
original_trace(E:(IsEventSpace?),C:setof[event]):
                                  [(Path(Program_Slice(E,C))) → trace] =
```

```
λ(tr:(Path(Program_Slice(E,C))))(n:nat):
 let etr = extended_trace(E,C)(tr) in
  if n=0 then tr(0) else
   let m = min(λ(x:nat): Card(etr(x)`h)>=n) in
    if Card(etr(m)`h)= n then etr(m) else
     let D = etr(m)`h \etr(m-1)`h, p = Card(etr(m-1)`h) in
      (# h := union(etr(m-1)`h,spell_history(E)(n-p,D)),
         σ := spell_store(E)(n-p,D)(etr(m-1)`σ) #)
    endif
  endif
```

The approach used to reconstruct traces (paths) on the original system from traces in the reduced system is general, so it can be extended to other state-space reduction techniques, provided that some *sufficient* conditions are verified. Note that the definition of `original_trace` depends on the proper definition of `extended_trace`. The three lemmas below summarize those sufficient conditions. The first lemma says that the *minimum* index `i` for which the cardinality of `extended_trace` is greater than or equal than `n`, for some `n`, always exists. The second lemma says that this cardinality is always positive for any positive `n`. And the third lemma says that `extended_trace` histories grow.

```
etr_nonempty_n_positive: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:(Path(Program_Slice(E,C))),n:nat):
  let S=λ(i:nat): Card(extended_trace(E,C)(tr)(i)`h) >= n in
   nonempty?[nat](S)

etr_min_positive: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:(Path(Program_Slice(E,C))),n:nat):
  let S=λ(i:nat): Card(extended_trace(E,C)(tr)(i)`h) >= n in
   n>0 ⇒ min(S)>0

etr_n_minus_p_nonnegative: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:(Path(Program_Slice(E,C))),n:nat):
  let S=λ(i:nat): Card(extended_trace(E,C)(tr)(i)`h) >= n in
   let m=min(S) in let p=Card(extended_trace(E,C)(tr)(m-1)`h) in
    n>0 ⇒ n-p>=0
```

Further, the lemmas below summarize some properties about original traces. The first lemma follows from the proper definition of `spell_history`; the second from the definition of the first lemma, and the third from the first lemma and some results on sets theory.

```
otr_makes_single_steps : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat):
  Path(Program_Slice(E,C))(tr) ⇒
  subset?(original_trace(E,C)(tr)(i)`h,original_trace(E,C)(tr)(i+1)`h) ∧
  singleton?(original_trace(E,C)(tr)(i+1)`h \original_trace(E,C)(tr)(i)`h)

otr_are_strict_subsets : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat) :
  Path(Program_Slice(E,C))(tr) ⇒
```

```
  strict_subset?(original_trace(E,C)(tr)(i)`h,
                 original_trace(E,C)(tr)(i+1)`h)

otr_as_increments_the : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat) :
  Path(Program_Slice(E,C))(tr) ⇒
  original_trace(E,C)(tr)(i+1)`h =
    add(the(original_trace(E,C)(tr)(i+1)`h \original_trace(E,C)(tr)(i)`h),
        original_trace(E,C)(tr)(i)`h)
```

Now, we go into the proof of the following theorem, which summarizes the process of constructing original traces described before.

```
constructing_original_traces_from_traces : theorem
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace):
  Path(Program_Slice(E,C))(tr) ⇒ Path(E)(original_trace(E,C)(tr))
```

Theorem 1 (`constructing_original_traces_from_traces`)**.** Because of the following equivalence:

```
∀(E:(IsEventSpace?),tr:trace):Path(E)(tr) ⇔ ∀(i:nat):PathUpTo(E)(tr)(i),
```

where `PathUpTo` is given by:

```
PathUpTo(E:(IsEventSpace?))(tr:trace)(n:nat) : RECURSIVE bool =
 if n=0 then InitialState(tr(0))
 else PathUpTo(E)(tr)(n-1) ∧ NextState(E)(tr(n-1),tr(n)) endif
measure n
```

the proof reduces to:

```
Path(Program_Slice(E,C))(tr) ⇒ ∀(i:nat): PathUpTo(E)(tr)(i)
```

Then, by induction on `i`, the base case becomes:

```
Path(Program_Slice(E,C))(tr) ⇒ PathUpTo(E)(tr)(0)
```

When expanding `Path` and `PathUpTo` definitions, the base case reduces to:

```
InitialState(tr(0)) ∧ ∀(i:nat): NextState(Program_Slice(E,C))(tr(i),
tr(i+1)) ⇒ InitialState(original_trace(E,C)(tr)(0))
```

Because `original_trace(E,C)(tr)(0)` is `tr(0)`, this goal reduces trivially. For the case `i=k` we have:

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k)
⇒ PathUpTo(E)(original_trace(E,C)(tr))(k+1)
```

Then, because `PathUpTo(E)(original_trace(E,C)(tr))(k+1)` can be expressed as the conjunction between `PathUpTo(E)(original_trace(E,C)(tr))(k)` and `NextState(E)(original_trace(E,C)(tr)(k), original_trace(E,C)(tr)(k+1))`, the case `i=k` reduces to:

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k)
⇒ NextState(E)(original_trace(E,C)(tr)(k),original_trace(E,C)(tr)(k+1))
```

When expanding the definition of `NextState`, the proof of the `i=k` reduces to three sub-cases, namely, (*ii.a*) which states that if `original_trace` makes a transition from state at index `k` to state at index `k+1` using `the` single event `e` in the difference between the state histories, then any event `f` occurring before `e` must belong to the history of `original_trace` at index `k`.

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k)⇒
 ∀(f:event):
 (carrier(E)(f) ∧
  E(f,the(original_trace(E,C)(tr)(k+1)‘h \original_trace(E,C)(tr)(k)‘h))∧
  f /= the(original_trace(E,C)(tr)(k+1)‘h \original_trace(E,C)(tr)(k)‘h)
 ) ⇒ original_trace(E,C)(tr)(k)‘h(f)
```

(*ii.b*) which states that for any indexes `k` and `k+1` in `original_trace`, the history at index `k+1` can be obtained from the history at index `k` when adding `the` event in the difference.

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k)⇒
original_trace(E,C)(tr)(k+1)‘h =
 add(the(original_trace(E,C)(tr)(k+1)‘h \original_trace(E,C)(tr)(k)‘h),
     original_trace(E,C)(tr)(k)‘h)
```

and (*ii.c*) which states something similar to (*ii.b*), but considering stores instead of histories:

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k)⇒
original_trace(E,C)(tr)(k+1)‘σ =
 cases the(original_trace(E,C)(tr)(k+1)‘h \original_trace(E,C)(tr)(k)‘h) of
  Write(t,l,r): original_trace(E,C)(tr)(k)‘σ with [l:=r]
 else original_trace(E,C)(tr)(k)‘σ endcases
```

We are not going into details about the proof of these three sub-cases here; we just want to say that the proof of (*ii.b*) is based on the correct definition of `spell_history`; (*ii.c*) on the correct definition of `spell_store`; and (*ii.a*) on the correct definition of both `spell_history` and `spell_store`.

Theorem 2 uses lemma `constructing_original_traces_from_traces` to prove `preserving_slice_if`.

Theorem 2 (`preserving_slice_if`). After expanding the definition of `Holds` and `Sem` and doing induction on `prop`, `preserving_slice_if` reduces to:

```
( ∀(s:state): InitialState(s) ⇒
   (∃(tr: trace): tr(0) = s ∧ Path(Program_Slice(E,C))(tr) ∧
                              ∀(j:nat): Eval(P)(tr(j)‘σ)) ) ⇒
( ∀(s:state): InitialState(s) ⇒
  (∃(tr: trace): tr(0) = s ∧ Path(E)(tr) ∧ ∀(j:nat): Eval(P)(tr(j)‘σ)) )
```

Then, when considering the same state `s` in the hypothesis as in the goal, the theorem reduces to:

```
( InitialState(s) ∧
  ∃(tr: trace):
```

```
   tr(0)=s ∧ Path(Program_Slice(E,C))(tr) ∧ (∀(j:nat): Eval(P)(tr(j)‘σ))
) ⇒
( ∃(tr: trace): tr(0) = s ∧ Path(E)(tr) ∧ (∀(j:nat): Eval(P)(tr(j)‘σ)) )
```

To instantiate the goal, `original_trace(E,C)(tr)` is used. Thereafter, three subgoals are to be proved, namely:

(*i.*) `tr(0)=s` ⇒ `original_trace(E,C)(tr)(0)=s`
(*ii.*) `Path(Program_Slice(E,C))(tr)`⇒`Path(E)(original_trace(E,C)(tr))`
(*iii.*) ∀`(j:nat): Eval(P)(tr(j)‘`σ) ⇒
∀`(j:nat): Eval(P)(original_trace(E,C)(tr)(j)‘`σ)

Since `original_trace(E,C)(tr)(0)` is `tr(0)`, (*i.*) is trivially verified; (*ii.*) reduces from Theorem 1; to prove (*iii.*), `original_traces_are_well_formed` below is used. This lemma says that for all index `j` in the original trace exists an index `i` in the sliced trace such that any event in the difference is unimportant, *i.e.* it does not modify the validity of the property that is checked. This last lemma constitutes a new *sufficient* condition.

```
original_traces_are_well_formed : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace) :
  Path(Program_Slice(E,C))(tr) ⇒
  ∀(j:nat):∃(i<=j): subset?(tr(i)‘h,original_trace(E,C)(tr)(j)‘h) ∧
                    ∀(b:event): (original_trace(E,C)(tr)(j)‘h \tr(i)‘h)(b)
                                ⇒ unimportant_event(C)(b)
```

6 Conclusion

The full PVS formalization presented here consists of 1800 lines of code, including 32 theorems. This formalization shows how theorem proving techniques can effectively be used to prove general properties about state-space reduction algorithms. We presented the formalization of a slicing algorithm introduced previously in [1], and the proof that this algorithm preserves a subset of the properties that can be modeled using Computation Tree Logic (CTL): *next-state* properties formed of `AX`, `EX` CTL operators are not preserved under slicing.

Furthermore, during the proof some *sufficient* conditions were outlined to extend the two-steps path reconstruction proof approach to other proofs that involve proving CTL property-preserving under similar state-space reduction techniques. In particular, in future work, we are interested in exploring how partial order reduction techniques [6] can be employed to reduce the number of states generated in `MDD`s based symbolic state generation techniques [3]. And we are interested in the correctness proofs involved in that reduction.

The Java Memory Model (JMM) as specified in Chapter 17 of the Java Language Specification presents some inconsistencies as highlighted by *W. Pugh* in [9]. A new document of Java Specification Requests (JSR-133) (see `http://-www.cs.umd.edu/~pugh/java/memoryModel/`) has been produced to fix these inconsistencies. This document is part of the most recent Tiger 5.0 release of

Java. We consider that main results presented here are still valid for these new specifications: our results apply not only for slicing techniques in the context of Java, but for reduction techniques in general.

Acknowledgements. We thank anonymous referees for useful feedback and Gerald Lüttgen for his comments on the previous of this paper. This work has been partially supported by the EPSRC under grant GR/S86211/01.

References

1. N. Cataño. Slicing event spaces: Towards a Java programs checking framework. In Thomas Arts and Wan Fokkink, editors, *FMICS: Eighth International Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Lecture Notes in Computer Science*, Trondheim, Norway, Jun. 5–7 2003. Elsevier.
2. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer-Verlag, 1999.
3. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 379–393, Warsaw, Poland, 2003. Springer-Verlag.
4. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Logics of Programs*, Lecture Notes in Computer Science, pages 52–71, Yorktown Heights, New York, May 1981.
5. E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of 14th Symposium on Theory of Computing (STOC'82)*, pages 169–180, San Francisco, CA, May 1982. ACM.
6. Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.
8. S. Owre, N. Shankar, J.M. Rushby, and D.W.J Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI, Menlo Park, CA, Nov. 2001.
9. W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 89–98. ACM Press, 1999.
10. B. Reus and T. Hein. Towards a machine-checked Java specification book. In *TPHOL, Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 480–497. Springer-Verlag, 2000.

Proving Equalities in a Commutative Ring Done Right in Coq

Benjamin Grégoire and Assia Mahboubi

INRIA Sophia-Antipolis, 2204, routes des Lucioles - B.P. 93,
06902 Sophia Antipolis Cedex, France
{Benjamin.Gregoire, Assia.Mahboubi}@sophia.inria.fr

Abstract. We present a new implementation of a reflexive tactic which solves equalities in a ring structure inside the Coq system. The efficiency is improved to a point that we can now prove equalities that were previously beyond reach. A special care has been taken to implement efficient algorithms while keeping the complexity of the correctness proofs low. This leads to a single tool, with a single implementation, which can be addressed for a ring or for a semi-ring, abstract or not, using the Leibniz equality or a setoid equality. This example shows that such reflective methods can be effectively used in symbolic computation.

1 Introduction

In the context of a computer algebra system, one of the most extensively used functionalities is the simplification of symbolic expressions, and in particular, the use of algebraic identities. These identities are usually established by elementary combinations of canonical identities, stored in a very large database, in a quite efficient way. Programing similar tools in a proof assistant consists in programing decision procedures, as the user is concerned with the reliability of the result.

Algebraic identities that the user of proof assistant is to handle are often equalities modulo the axioms of a ring. There are numerous examples of such identities: the product of two bi-squares is itself a bi-square, remarkable identities like the famous $(a+b)^2 = a^2 + 2ab + b^2$ or event more complex properties like the fact that the product of sums of eight squares is a sum of eight squares. These equalities are decidable and it seems natural to relieve the user of a proof assistant of such goals, by providing an automatic tool. Otherwise the proof of the identity:

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

would require no more than thirty elementary rewriting steps of the ring axioms.

The Coq [12] proof assistant already provides such a tool called `ring`. It is not based on an automatic rewriting strategy but built using a reflexive technique [3]. The use of reflexivity has already reduced the size of the generated proof terms and the time for building and checking them. Nevertheless, the efficiency of `ring` is not satisfactory. For example, proving $10 * 100 = 1000$, is immediate if the multiplication ranges over the integers, while it takes about a

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 98–113, 2005.

hundred seconds on a 3GHz machine if the multiplication ranges over the axiomatic implementation of real numbers. The efficiency of the method on such goals should not depend on the computational nature of the underlying ring structure. This bad behaviour on constants strongly affects the efficiency of the method on algebraic identities of higher degree. Moreover the implementation choices made in the `ring` developpment are really limiting the size of the entries `ring` is able to deal with.

Currently, there exists eight different implementations of `ring` depending on the kind of ring: semi-ring or ring, abstract or not, setoid equality or Leibniz equality. Here, we factorize these eight implementations through a modular implementation which will be finally instantiated to fit the kind of ring required.

The Coq system has recently been improved by the introduction of a compiler and an abstract machine, which now allows the evaluation of Coq programs with the same efficiency as Ocaml programs [8]. After the experiences of marrying computer algebra systems with theorem provers to get both efficiency and reliability [9], it now seems reasonable to use Coq as a single environment for programming, certifying and evaluating computer algebra algorithms. Our `newring` decision procedure is one of these efficient tools required for the manipulation of symbolic expressions, showing that the reflexive methods are the way to separate computations from checking, inside the proof assistant. Furthermore it is the first step for a bunch of other decision procedures, like the simplification of field equalities [6], or decision methods in geometry [11].

In Section 2, we begin with some general remarks about the reflexive method and its use in our particular context. The Section 3 is dedicated to our choice to get efficient representation of polynomials, which is a crucial point for the efficiency. The Section 4 shows the major importance of the choice of coefficients set for these polynomials. In the Section 5, we introduce a new axiomatic structure, called *almost-ring*, which allows to unify the implementations of the procedure for rings and semi-rings. In Section 6 we show how the use of the new metalanguage Ltac [5,2] allows to completely avoid the use of external Ocaml code. Section 7 is dedicated to examples and benchmarks before we conclude in Section 8.

2 Overall View of the Method

2.1 Reflexivity

In the Coq system, the rewriting steps are explicit in a proof: each step builds a predicate having the size of the current goal when the rewriting was performed, hence the size of the proof term heavily depends on the number of these rewriting steps. The reflection technique introduced by [1] takes benefit of the reduction system of the proof assistant to reduce the size of the proof term computed and consequently to speed up its checking. It relies on the following remark:

– Let $P : A \to Prop$ be a predicate over a set A.

– Suppose that we are able to write in the system a semi decision procedure f, such that f is computable and if f returns *true* on the entry x, then $P(x)$ is valid, that is to say:
`f_correct: forall x, f(x)=true -> P(x).`

If we want to prove $P(y)$ for a particular y, and if we know that $f(y)$ *reduces* to *true*, then we can simply apply the lemma `f_correct` to y and to a proof that $true = true$. Thanks to the conversion rule which allows to change implicitly the type of a term by an equivalent (modulo β-reduction):

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \equiv U}{\Gamma \vdash t : U}$$

This latter proof, which is `(refl_equal true)`, is also implicitly a proof that $f(y) = true$ because $f(y)$ reduces to *true*, so $true = true$ is convertible with $f(y) = true$. Finally the proof of $P(y)$ we have built is :

```
f_correct y (refl_equal true)
```

The size of such a proof now only depends on the size of the particular argument y and does not depend on the number of implicit β-reduction steps: explicit rewriting steps have been replaced by implicit β-reductions. The size of the proof term of the correctness lemma for f may be large, it is only done once and for all. It will be shared by all the instantiations and will no more be type-checked. The efficiency of this technique of course strongly depends on the efficiency of the system to reduce the application of the decision procedure $f(y)$, hence on the efficiency of the decision procedure itself.

2.2 General Scheme of the `newring` Tactic

The `newring` tactic operates on a ring structure A, which includes a base type for its elements, two constants 0 and 1, three binary operations $+, *, -$ over A and an opposite unary function $-$, together with the usual axioms defining a commutative ring structure. Its goal is to prove the equality of two terms t_1 and t_2 of type A modulo the ring axioms.

Working by reflection means that we want to build a semi decision procedure f, which will take t_1 and t_2 as arguments and return *true* if t_1 and t_2 are equal modulo associative-commutative rewriting in the ring structure.

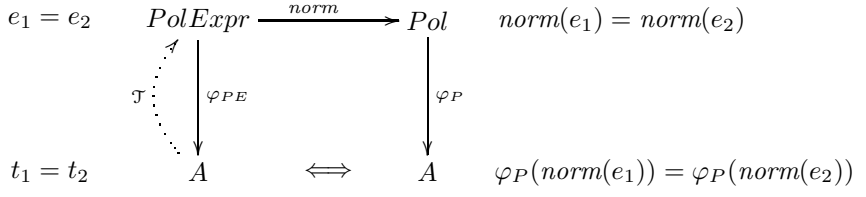
A natural way to perform a comparison between two terms seems to be the pattern-matching. Yet the Coq system does not allow pattern matching over arbitrary terms, but only over inductive types. That is why terms of the type A are going to be *reflected* into an appropriate inductive type $PolExpr$, which describes the syntax of terms of type A. This step is also called the *metaification*. A term of type A is mapped by the meta-function $\mathcal{T}$ to a polynomial expression in $PolExpr$ by:

– interpreting every ring constant as a constant polynomial expression (eg. 0,1)
– interpreting every ring operation as an operation over polynomial expressions

– hiding every subterm which is neither a ring constant, nor the application of a ring operation to other subterms behind a labeled variable and building the corresponding association list.

$\mathcal{T}$ is a kind of oracle, we will explain in Section 6 how to build such a function using the *meta*-language Ltac[5] which allows to do pattern-matching over an arbitrary Coq expression.

Once we have built the two *PolExpr*, e_1 and e_2, corresponding to t_1 and t_2, the idea is to check the equality of the normal forms of e_1 and e_2 and to prove that this implies the equality of t_1 and t_2. For this purpose, we should ensure the correctness of the following diagram:



by the correctness lemma:

$$\forall e \in PolExpr,\ \varphi_{PE}(e) = \varphi_P(norm(e))$$

φ_{PE} (resp. φ_P) are the evaluation functions. They evaluate polynomial expressions (resp. normalized polynomial *Pol*) into elements of A, by interpreting back each constant polynomial to a constant of A, each variable by the ring term it was hiding and each representation of an operator by the corresponding ring operator.

These functions can be easily defined within the theory by pattern matching over the reflected inductive types.

The inductive type *PolExpr* is adapted to the metaification. To ensure the completeness of our tactic it should verify the following meta property:

$$\forall a \in A.\ \varphi_{PE}(\mathcal{T}(a)) = a.$$

Note that we do not have to prove this property, which can not be expressed inside Coq. It does not affect the correctness of our decision procedure, but only its completeness.

The type *Pol* stands for the set of the normalized forms of polynomial expressions, which does not need to be the same as *PolExpr*. It is adapted to build normal forms efficiently. The *norm* function bridges the gap between these two kind of constraints: *PolExpr* suits to the syntax of the terms in A and *Pol* allows efficient computations.

To prove the equality of t_1 and t_2, our tactic first computes e_1 and e_2 using $\mathcal{T}$, and then checks the equality of their normal forms. If it holds, the correctness lemma and the transitivity of equality ensure the equality of t_1 and t_2:

$$t_1 = \varphi_{PE}(\mathcal{T}(t_1)) = \varphi_P(norm(\mathcal{T}(t_1))) = \varphi_P(norm(\mathcal{T}(t_2))) = \varphi_{PE}(\mathcal{T}(t_2)) = t_2$$

3 Sparse Horner Normal Forms

Choosing the shape of the normal form is a crucial point for the complexity. The normal form for terms in the ring will be determined by the choice made for the normal form of polynomial expressions. We present here the choice we made for the normal form, the sparse Horner normal form, which provides the required efficiency.

3.1 Representation

Horner form for polynomials in $C[X]$ can be represented by the following inductive type:

```
Inductive Pol1 (C:Set) : Set :=
 | Pc : C -> Pol1 C
 | PX : Pol1 C -> C -> Pol1 C.
```

where `(Pc c)` represents the constant polynomial c and `(PX P c)` represents the polynomial $P * X + c$. The problem with such a representation is that a polynomial can have a lot of holes due to gaps in the degrees. For example, $X^4 + 1$ is represented in the Horner form as:
`(PX (PX (PX (PX (Pc 1) 0) 0) 0) 1)`. The number of nested `PX` constructors of such a polynomial is indeed its degree. To get a more compact representation of the Horner form we can factorize these gaps by adding a power index in the constructor of non constant polynomials:

```
Inductive Pol1 (C:Set) : Set :=
 | Pc : C -> Pol1 C
 | PX : Pol1 C -> positive -> C -> Pol1 C.
```

where `positive` is a inductive type representing $\mathbb{N}^*$.

Now `(PX P i c)` stands for the polynomial $P * X^i + c$. So $X^4 + 1$ is now represented as `(PX (Pc 1) 4 1)`.

Once the representation of univariate polynomials is fixed, there is a natural way to extend it to multivariate polynomials, using the canonical isomorphism $C[X_1, \ldots, X_n] = C[X_1 \ldots X_{n-1}][X_n]$. In Coq this can be done by declaring the following fixpoint using dependent type:

```
Fixpoint Poln (C:Set) (n:nat) {struct n} : Set :=
 match n with
 | 0 => C
 | S m => Pol1 (Poln C m)
 end.
```

The type `(Poln C n)` represents the set of polynomials with n variables. Namely `(Poln C (S n))` represents the set of univariate polynomials with coefficients in `(Poln C n)` and `(Poln C 0)` is the set of constant polynomials in C.

This representation creates another kind of holes corresponding to holes in variables. For example the polynomial 1 will be encoded either by `(Pc 1)` if it

is seen as an element of $Z[X]$ or by `(Pc (Pc (Pc (Pc 1))))` if it is seen as an element of $Z[W, X, Y, Z]$. To solve this problem, we give up the idea of defining multivariate polynomials recursively from univariate ones. We now define the set of polynomials in an arbitrary number of variables in one shot.

```
Inductive Pol (C:Set) : Set :=
 | Pc : C -> Pol C
 | Pinj : positive -> Pol C -> Pol C
 | PX : Pol C -> positive -> Pol C -> Pol C.
```

- `(Pc c)` stands for the constant polynomial $c \in C[X_1, \ldots, X_n]$ for any n.
- If $Q \in C[X_1, \ldots, X_{n-j}]$, and `Q` is its representation, then `(Pinj j Q)` represents Q as a polynomial in n variables, namely $Q.X^0_{n-j+1} * \ldots * X^0_n$. We have "pushed" Q from $C[X_1, \ldots, X_{n-j}]$ to $C[X_1, \ldots, X_n]$. j is called the injection index.
- Finally, `(PX P i Q)` stands for $P * X^i_n + Q$ where $P \in C[X_1 \ldots X_n]$ and $Q \in C[X_1 \ldots X_{n-1}]$ is constant in X_n.

3.2 Normalization

Our sparse Horner form does not provide a unique representation for arbitrary polynomials. In $C[X]$ the polynomial $X^4 + 1$ can be represented by `(PX (Pc 1) 4 (Pc 1))` or by `(PX (PX (Pc 1) 3 (Pc 0)) 1 (Pc 1))`. To solve this, we can define a normalization function that build a canonical representative of a polynomial, and then define the equality on polynomial as the equality of the canonical representatives.

Instead of normalizing before checking equality, our choice is to always manipulate canonical representatives verifying the three following properties:

- the coefficient of highest degree is never zero;
- the injection index is the biggest possible;
- the power index is the biggest possible.

So the canonical representative of $X^4 + 1$ is `(PX (Pc 1) 4 (Pc 1))`. Note that, it is also the most compact representation of a sparse Horner form. Since the complexity of operations depends on the size of the polynomials, linear for addition and quadratic for multiplication, it is interesting to work with canonical terms. This means that each operation on polynomials should only build canonical terms. If `P` and `Q` are in canonical form, building the canonical representation of `(PX P i Q)` is not expensive, since we only need to locally destruct `P`:

- if `P = (Pc 0)` then build the canonical representative of `(Pinj 1 Q)`;
- if `P = PX P' i' (Pc 0)` then the canonical representative is:
  `(PX P' (i+i') Q)`
- else `(PX P i Q)` is the canonical representative.

Our defined operations on polynomial, denoted by `Padd`, `Pmul`, `Psub`, and `Popp`, keep the following invariant: if their arguments are canonical then their

result is canonical. To ensure this, we use specialized constructors that perform local normalizations: `mkPinj` and `mkPX`. For example, the addition of `(PX P i Q)` and `(PX P' i Q')` leads to the term `(mkPX (Padd P P') i (Padd Q Q'))`. Since the addition of `P` and `P'` can be the zero polynomial, we need to use `mkPX` to ensure that the result is canonical. But we directly use constructors `Pinj` and `PX`, which are costless, each time the invariant allows it, as in the addition of `(PX P i Q)` and `(Pc c)` which reduces to `(PX P i (Padd Q (Pc c)))`, here `P` can not be zero or of the form `(PX P' i' (Pc 0))`, since `(PX P i Q)` is canonical, so `(PX P i (Padd Q (Pc c)))` is canonical.

For each operator, we prove a correctness lemma showing that the operator is correct up to evaluation. For the addition the lemma is:

```
Lemma Padd_correct: forall P Q l,
     phiP l (Padd P Q) == (phiP l P) + (phiP l Q).
```

where `==` is the setoid equality over the initial ring (or semi-ring) structure and `+` is its addition.

Note that using `mkPX` instead of `PX` has no influence on the correctness, because `(phiP l (mkPX P i Q))` is equal to `(phiP l (PX P i Q))`. The only influence is for completeness, since using `PX` instead of `mkPX` can produce a non-canonical representative. But again, we do not need to prove completeness.

The normalization function from polynomial expressions to their canonical sparse Horner forms consists in mapping variables to monomials, constants to constant polynomials and operation constructors to operation functions on Horner form. The canonical representative is given by the evaluation of the term obtained.

After having defined the normalization function, we can prove its correctness:

```
Lemma norm_correct : forall l e, phiPE l e == phiP l (norm e).
```

And then the main lemma, which expresses the correctness of our decision procedure:

```
Lemma f_correct : forall l e1 e2,
  Peq (norm e1) (norm e2) = true -> phiPE l e1 == phiPE l e2.
```

where `Peq` stands for a defined function which checks the syntactic equality over sparse Horner forms.

The set of coefficients C is the carrier of the computations performed by the normalization function. The following section will show that the choice made for C is crucial, especially for the efficiency of the procedure, as C catches the "best computational part" of the ring.

4 Computations over the Parametric Coefficient Set

The normalization function we have described above strongly relies on the computational behavior of the set of coefficients. For example the normalization of

$x + (-x)$ leads to $(1 + (-1)).x$, which will *reduce* to $0.x$. C has to be chosen as a set over which we know how to compute, as efficiently as possible. In the Coq system, these kind of sets will be represented by inductive types, and the operations are defined as functional programs.

In the Coq system, `Z` is an implementation of $\mathbb{Z}$ as lists of binary digits. In the case `Z` is the underlying ring of the equality to be proved, `Z` itself is a good candidate. On the other hand, if the underlying ring is `R`, the axiomatic implementation of real numbers in Coq, `R` itself will not be an appropriate set of coefficients. Indeed, in `R`, $1 + (-1)$ is equal to 0 (using ring axioms) but does not reduce to 0: the subtraction as the other operations and constants of `R` are only symbols, and are not evaluable. Hence $x + (-x)$ would not reduced to $0.x$ by the normalization function. Since there is a natural inclusion of $\mathbb{Z}$ in $\mathbb{R}$, we can use `Z` as a set of coefficients. Moreover, whatever ring A we are dealing with, the canonical morphism from $\mathbb{Z}$ to A will enable us to use again `Z` as a set of coefficients. This type `Z` seems then to be a universal candidate for coefficients.

Nevertheless, `Z` will not always be the good choice. If the computational content of the ring operations is stronger than the ring axioms, this method will allow to prove more than what is provable by sole rewriting of the rings axioms. In the case we are working in the ring `bool`, the equality $x + x = 0$ holds, even if it is not provable using only the ring axioms. The good choice for C is now `bool` itself: the left side of the equality is again reflected in $X + X$ (with coefficients in `bool`), whose normal form $(1 + 1).X$ is reduced to $0.X = 0$ by the normalization function, thanks to the computations over the coefficients in `bool`. Hence our choice is to parametrize our tactic by the set of coefficients and to let the user make the most appropriate choice.

An inductive type has to fulfill some requirements to be admissible as a set of coefficients. These requirements will ensure the correctness of the normalization function. Formally, C will be admissible if it is equipped with the constants and operators of a ring, and with a decidable equality relation $=_C$. The last requirement is needed to implement the `mkPX` and `mkInj` constructors (we need to be able to check the equality at 0). It also allows to get a decidable equality on sparse Horner form.

We also require a suitable evaluation function from C to A, mapping the constants of C to the elements of A and this function should be compatible with the respective operations of C and A. These requirements can be expressed by the existence a so-called *morphism* between C and A (even if C does not need to be a ring). This morphism evaluates the constants and operators in C into their analogous in A, and the decidable equality relation $=_C$ over C should satisfies : if $(x =_C y)$ returns *true*, then the evaluations of x and y will be equal in A.

Once we have got C and a proof of all these specifications, we define in a generic way the operations over polynomials as explained in Section 3, and extend the morphism between C and A into two evaluation functions φ_{PE} and φ_P, from the polynomial expressions and sparse Horner form to A. We also obtain a proof of the general diagram of the reflection presented in 2.2, *Pol*

and *PolExpr* being now replaced by their parametrized version $Pol(C)$ and $PolExpr(C)$.

We have implemented the identity morphism which corresponds to taking the ring itself as the set of coefficient. The user can always apply the resulting tactic even if it may not prove much equalities (like in the case `R` is involved). We have also implemented the morphism from `Z` to an arbitrary ring, which can always be used as an efficient default choice, but is not necessary the best choice (cf the case of `bool`).

In order to get the maximal efficiency from this method, the user has to make to most appropriate choice for C. If the ring structure is defined in an axiomatic way, like `R`, `Z` will always be a good choice for the set of coefficients. In the case the ring already presents a computational content, like `Z` or `bool`, it may be a good choice to take the ring itself as the coefficient set. Nevertheless, if the available operations are not efficient enough, like it is the case for example in the semi-ring of Peano numbers, it may be more appropriate to obtain the most efficient computational content by changing the set of coefficients all the same, here for example by taking a binary representation of natural numbers.

5 Unifying Rings and Semi-rings

A semi-ring is a ring where the axioms stating the existence of an opposite (and of a subtraction) have been replaced by an extra axiom : $\forall x, 0 * x = 0$. These structures are quite alike and we would like to get a tool also adapted to semi-rings without duplicating the code. For this purpose, we work with an intermediate structure, called *almost-ring*. The idea is to complete a semi-ring with a unary operator, called *almost-opposite* which is morally the opposite operator of a ring structure. This operator will be instantiated by a dummy function to equip a semi-ring with such a structure. In fact the fundamental remark is the following : in the correctness proof of the normalization function, the axiom defining the *opposite* operator as an inverse, by stating that $\forall x, x + (-x) = 0$ is never used itself, but only the properties which describe its combination with the other operators. Finally an almost-ring is defined by the following axioms:

- $\forall x,\ 0 + x = x$
- $\forall x\ y,\ x + y = y + x$
- $\forall x\ y\ z,\ x + (y + z) = (x + y) + z$
- $\forall x,\ 1 * x = x$
- $\forall x\ y,\ x * y = y * x$
- $\forall x\ y\ z,\ x * (y * z) = (x * y) * z$
- $\forall x\ y\ z,\ (x + y) * z = x * z + y * z$
- $\forall x, 0 * x = x$ (at that point we have a semi-ring)
- $\forall x\ y,\ -(x * y) = -x * y$ (combination of pseudo-opposite with product)
- $\forall x\ y,\ -(x + y) = -x + -y$ (combination of pseudo-opposite with addition)
- $\forall x\ y,\ x - y = x + -y$ (definition of an associated pseudo-subtraction)

It is straightforward to prove that every ring is an almost-ring. The axioms of an almost-ring do not allow to prove the missing axiom defining the opposite in ring $x + -x = 0$. Anyway, this identity will be proved by our tactic, provided that in the set of coefficients $1 + (-1)$ reduces to 0. This is ensured thanks to the existence of a morphism from the set of coefficients to the ring. Every semi-ring can also be equipped with an almost-ring structure if we take the identity as an almost-opposite operator and the defined addition operator of the semi-ring as subtraction.

The tactic is finally designed for an almost-ring structure. We have moreover built the proofs required to transform any ring or semi-ring into the associated almost-ring.

The last parameter given to the tactic is the equality relation used over the ring. It may not be the Leibniz equality, but an equivalence relation adapted to the ring structure. For example, this is the case for an implementation of $\mathbb{Q}$ as $\mathbb{Z} \times \mathbb{N}^*$. A set equipped with such an equality relation is called a setoid ([7],[10]). Proving equalities in such a setoid ring requires extra properties stating that all the ring operations are compatibles with the given setoid equality. In the case the equality involved in the goal is the Leibniz one, these requirements are trivial to fulfill. That is why the tactic will finally also be parametrized by a setoid equality and the related compatibility lemmas for the operations.

6 Programming the Metaification and the Tactic

The purpose of the `newring` tactic is to solve goals of the form $t_1 == t_2$ by applying the `f_correct` lemma. To do so we need to produce a list of values l and two polynomial expressions e_1 and e_2 such that the evaluation of e_1 (resp. e_2) at l is convertible to t_1 (resp. t_2). Consider the following equality

$$3 * \sin(x) * x = x * (\sin(x) + 2 * \sin(x)) + 0 * y$$

In this case l will be $[\sin(x);\ x;\ y]$, e_1 will be $3 * X_1 * X_2$ and e_2 will be $X_2 * (X_1 + 2 * X_1) + 0 * X_3$.

6.1 Programming the Metaification

We use the Coq proof-dedicated metalanguage Ltac[5] to design the oracle producing the expected values (l, e_1, e_2). This metalanguage allows to do pattern-matching on arbitrary Coq terms, and thereby to program this metafunction, which is a tactic, in a natural way as done in [6].

We first build a function `FV` which computes the list l containing the subterms to abstract. These are the ones which do not belong to the syntax of a ring. Then the `mkPolexpr` tactic computes the two expresssions e_1 and e_2 and the list l is used to know which variable is associated to a given subexpression to abstract.

```
Ltac mkPolexpr Cst add mul sub opp t l :=
 let rec mkP t :=
```

```
  match t with
  | (add ?t1 ?t2) =>
    let e1 := mkP t1 in
    let e2 := mkP t2 in constr:(PEadd e1 e2)
  | (mul ?t1 ?t2) => ...
  | (sub ?t1 ?t2) => ...
  | (opp ?t1) => ...
  | _ =>
    match Cst t with
    | false => let p := Find_at t l in constr:(PEX p)
    | ?c => constr:(PEc c)
    end
  end
 in mkP t.
```

The tactic `mkPolexpr` takes as arguments a term `t`, the list `l` of terms to abstract, the ring operators and a tactic `Cst`. It matches the head symbol of `t`:

- If this symbol is one of the given operators then it builds recursively the corresponding polynomial expression;
- If the head symbol is not an operator then either `t` is a constant or it has to be abstracted into a variable. This discrimination is performed by the tactic `Cst` given in argument :
  - If `Cst` returns `false` then the index of the proper variable is given by the position of `t` in the list `l` given in argument.
  - Otherwise `t` is mapped to the corresponding constant.

The definition of the `Cst` tactic depends on the ring A. If A is an abstract ring, the set of coefficients will be `Z`, and we can already define a naive tactic which matches only the neutral elements of A (`r0` and `rI`).

```
Ltac genCstZ r0 rI t :=
  match t with
   | r0 => constr:(0%Z)
   | rI => constr:(1%Z)
   | _ => constr:false
   end.
```

On the other hand, in the case A is `Z`, the set of coefficients will be `Z` itself, and we can match much more constants: in fact all the terms built only with the constructors of `Z`.

```
Ltac ZCst t :=
 match (is_ZCst t) with
 | true => constr:t
 | false => constr:false
 end.
```

Here `is_ZCst` is a tactic matching the terms built only with the constructors of the inductive type `Z`.

This method has also been generalized to the case of semi-rings, where `N`, the implementation of binary natural numbers plays the role of `Z`. We have also built such a tactic `Cst` for boolean, where the target constants are booleans.

6.2 The Generic Tactic

To define the `newring` tactic itself, we use the possibility given by Ltac to program a higher-order function, which builds a tactic, solving equalities in the structure given in argument. For the sake of clarity we present a simplified version that can be used only if the goal is a valid equality modulo ring axioms and fails otherwise. The real implementation also replace both members of the equality by their normal form if they are not equal.

```
Ltac Make_ring_tac add mul sub opp req Cst_tac :=
  match goal with
  | [ |- req ?r1 ?r2 ] =>
    let fv := FV Cst_tac add mul sub opp (add r1 r2) (nil R) in
    let e1 := mkPolexpr Cst_tac add mul sub opp r1 fv in
    let e2 := mkPolexpr Cst_tac add mul sub opp r2 fv in
    apply (f_correct fv e1 e2); compute; exact (refl_equal true)
  | _ => fail "not equality"
  end.
```

The tactic first checks that the current goal is an equality. If so, it computes a single list `fv` of subterms to be abstracted in both terms, and the two polynomial expressions `e1` and `e2` representing the members of the equality. Then the tactic applies the correctness lemma `f_correct`. At that point the tactic should prove the hypothesis of the lemma, namely check that `(norm e1) ?== (norm e2)` is equal to `true`.

If `r1` and `r2` are equal modulo ring axioms then this new goal is convertible to `true = true`. So it is now possible to complete the proof with the term `(refl_equal true)`. The tactic `exact` checks that the provided term has a type convertible to the current goal `((norm e1) ?== (norm e2)) = true`. This is performed using a lazy reduction strategy. Here checking the convertibility is equivalent to computing the normal form of the equality's left-hand side. The efficient strategy suitable to this problem is the call by value reduction. So the tactic first uses the `compute` tactic to reduce the goal in this way, before concluding with `exact`.

We can now apply the `Make_ring_tac` to obtain a tactic which automatically prove ring equality in `Z`:

```
Ltac zring := Make_ring_tac Zplus Zmult Zminus Zopp (@eq Z) ZCst.
```

We also have implemented such a tactic for booleans (`bring`), reals (`rring`) and natural numbers (`nring`), Peano numbers as well as their binary implementation.

Finally, the `newring` tactic analyzes the type of the equality to prove and calls the corresponding specialized tactic:

```
Ltac newring :=
 match goal with
 | [|- @eq Z    _ _ ] => zring
 | [|- @eq R    _ _ ] => rring
 | [|- @eq bool _ _ ] => bring
 | [|- @eq nat  _ _ ] => nring
 end.
```

To work with an other user-defined structure, one can always use the predefined tactic `Make_ring_tac` to build the appropriate tactic for proving equalities in this structure.

7 Examples and Benchmarks

The `newring` tactic has performed two orthogonal improvements compared to the choices made in the `ring` tactic developped by S. Boutin [3]. The first one is the choice of the sparse Horner form for the representation of normal forms instead of an ordered sum of monomials, being themselves an ordered product of variables. The second is to use `Z` as the set of coefficients for reflected expressions when working with abstract rings (`R` for example).

7.1 Sparse Horner Form

Figure 1 describes the time to normalize the expression $(x_1 + \ldots + x_n)^d$ seen as a polynomial with coefficients in `Z`. For `ring`, the normal form of this expression is its expansion in an ordered sum of monomials, each prefixed by a coefficient in `Z`. Both tactics use `Z` as a set of coefficients, so these benchmarks show the interest of the sparse Horner form to deal with polynomials of higher degree. The gain in time for $n = 5$ and $d = 5$ is a factor 6 and a factor 500 for $n = 7$ and $d = 9$, thanks to the compactness of sparse Horner form representation. Using a naive Horner form (without power and injection index, or not maintaining canonical representatives) introduces an overhead of 30%. Moreover, the `ring` tactic is not able to normalize this expression when $n = 8$ and $d = 9$, and when $n = 12$ it fails for $d = 6$. The `newring` tactic is able to normalize the expression for $n = 12$ and $d = 11$.

Comparing the time to normalize expressions of the form $(x_1 + \ldots + x_n)^d$ to the results given by the `expand` function of Maple, is deceiving. The algorithm used by the computer algebra system in mainly focused on the access to a database of stored identities, and possible simple combinations of them. When the precomputed identities are useless, the system is of course less efficient, and can even fail because of the size of the normal form. This is the case for expressions of the following form

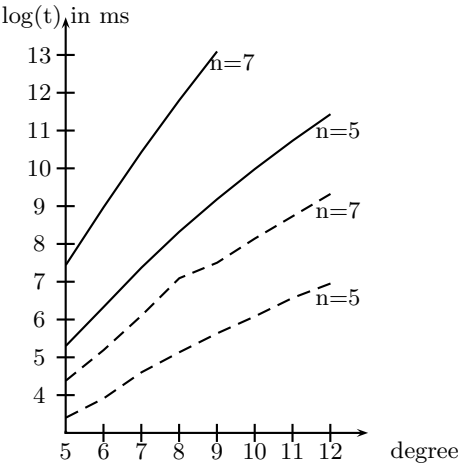


Fig. 1. Time to prove that $(x_1 + \ldots + x_n)^d$ is equal to its normal form

$$
\begin{array}{l}
(\,y \;+\; x_2 \;+\; \ldots \;+\; x_{n-1} \;+\; x_n)\,* \\
(x_1 \;+\; y \;+\; \ldots \;+\; x_{n-1} \;+\; x_n)\,* \\
\qquad\qquad\vdots \\
(x_1 \;+\; x_2 \;+\; \ldots \;+\; y \;+\; x_n)\,* \\
(x_1 \;+\; x_2 \;+\; \ldots \;+\; x_{n-1} \;+\; y\,)
\end{array}
$$

For $n = 8$ the `newring` tactic is four times slower than the `expand` strategy of Maple (0.4s for `newring`, 0.12s for Maple). But Maple fails to expand the formula when $n = 9$ (`Error, (in expand/bigprod) object too large`), while `newring` finishes in 1.7s.

7.2 The Set of Coefficients

Beside the successful use of the Horner form, the use of `Z` as the set of coefficients when we are working with an abstract ring has been a major improvement for efficiency. For the previous `ring` tactic, the representation of normal forms in an abstract ring leads to coefficients equivalent to unary numbers, hence computations are completely inefficient. Proving that 10*100 = 1000 takes about one hundred of seconds on a 3GHz machine using `ring`, and it is now immediate with `newring` (as one would expect). It is worth paying attention to the efficiency of such a tactic over (large) integers. One often deals with expressions with small coefficients but successive computations may increase their size in a significant way. A well-known phenomenon of explosion in the size of the coefficients occur while computing a remainder sequence of polynomials, like the computation of a polynomial gcd in $\mathbb{Q}[X]$. For example, in the context of the checking of computations made by an external oracle [9] (Maple or any dedicated program producing a trace of certificates...), checking the successive steps of such a com-

putation will force to deal with large coefficients, even if the initial polynomial entries had small ones.

8 Conclusion

This development shows that it is worth paying attention to the algorithmic aspects in programming such a procedure in the same way we would have done while programming it in a functional language. The choices we made in that sense turned out to be primordial for efficiency. This gain in efficiency could have lead to a complication of the associated correctness proofs. This is not the case, as the possible difficulties in the proofs lie in the mathematical complexity of the problem more than in the choices made for computations. This effort has even allowed to reduce the size of the development, by factorizing the eight versions of the tactic in a single one.

One other characteristic feature of the reflexive method is that it requires, for the reflection step, the use of an operator defined in the meta level, and hence using the meta-language of the system. The Ltac metalanguage turns out to be exactly the tool needed in reflexive tactic to program this reflection step in the meta-theory. The mechanism of pattern-matching over Coq terms indeed enables to write this function easily, without any knowledge of the inside of Coq and to work entirely at the top-level, without needing to compile again and again the whole sources of the system to integrate the new tactic.

A possible improvement for our development would be to allow negative powers in the representation of polynomials, to deal with Laurent series. But, one can also use the remark that proving an equality in a field can be transformed into a goal in a certain ring plus nonzero conditions for the denominators. This implementation of a `newfield` tactic has been achieved by L. Théry.

This work shows that the sparse Horner form is the right representation to compute efficiently with polynomials. We hope that existing developments, such as the decision procedure for geometry [11], strongly relying on the `ring` tactic will gain in efficiency and hence in power.

We are also convinced that this will allow the development of other efficient procedures to deal with symbolic expressions, providing a basic toolkit for larger developments in the domain of certified computer algebra. In particular, the second author uses the Horner representation of polynomials to develop a decision procedure for real numbers theory based on G. Collins' cylindrical algebraic decomposition [4], which is a quite complex algorithm resting on numerous computations over polynomials (computations of gcd, subresultant coefficients,...).

The efficiency of `newring` overcomes what was before a strongly limiting factor in such a development, showing that it is possible to compute efficiently within a proof assistant. This makes possible to use the proof assistant as a single environment for computing and proving as well as an efficient checker efficiently computations possibly performed by an external tool as described in [9].

The systematic use of `Z` as a set of coefficients has considerably increased the efficiency of the tactic. Yet `Z`, in which numbers are represented as lists of bits,

is not the best possible implementation for integers. An other step toward the efficiency of a genuine computer algebra system will be to provide to the user the possibility to use a library of machine binary integers, comprising fast computing operations, in order to deal even more efficiently with the huge integers occurring during symbolic computations (eg. polynomial gcds, prime numbers).

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 95–197, Philadelphia, Pennsylvania, June 1990. IEEE, IEEE Computer Society Press.
2. Y. Bertot and P.Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. 2004.
3. S. Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, pages 515–529, 1997.
4. G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. volume 33 of *Lecture Notes In Computer Science*, pages 134–183. Springer-Verlag, Berlin, 1975.
5. D. Delahaye. A Tactic Language for the System Coq. In *LPAR, Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000. http://cedric.cnam.fr/ delahaye/publications/LPAR2000-ltac.ps.gz.
6. D. Delahaye and M. Mayero. `Field`: une procédure de décision pour les nombres réels en Coq. In *Journees Francophones des Langages Applicatifs, Pontarlier*. INRIA, Janvier 2001.
   http://cedric.cnam.fr/ delahaye/publications/JFLA2000-Field.ps.gz.
7. V. C. G. Barthe and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, March 2003.
8. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.
9. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
10. M. Hofmann. A simple model for quotient types. In *TLCA '95*, pages 216–234, April 1995.
11. J. Narboux. A decision procedure for geometry in coq. In *TPHOLs*, pages 225–240, 2004.
12. The Coq development team. The coq proof assistant reference manual v7.2. Technical Report 255, INRIA, France, mars 2002. http://coq.inria.fr/doc8/main.html.

A HOL Theory of Euclidean Space

John Harrison

Intel Corporation, JF1-13, Hillsboro OR 97124
johnh@ichips.intel.com

Abstract. We describe a formalization of the elementary algebra, topology and analysis of finite-dimensional Euclidean space in the HOL Light theorem prover. (Euclidean space is $\mathbb{R}^N$ with the usual notion of distance.) A notable feature is that the HOL type system is used to encode the dimension N in a simple and useful way, even though HOL does not permit dependent types. In the resulting theory the HOL type system, far from getting in the way, naturally imposes the correct dimensional constraints, e.g. checking compatibility in matrix multiplication. Among the interesting later developments of the theory are a partial decision procedure for the theory of vector spaces (based on a more general algorithm due to Solovay) and a formal proof of various classic theorems of topology and analysis for arbitrary N-dimensional Euclidean space, e.g. Brouwer's fixpoint theorem and the differentiability of inverse functions.[1]

1 The Problem with $\mathbb{R}^N$

Since the pioneering work of Jutting [9], several people including the present author [6] have used computer theorem provers to formalize the construction of the real numbers and/or the development of elementary real analysis. There has also been some work in formalizing complex analysis, with proofs of the Fundamental Theorem of Algebra in Mizar [12], HOL Light [7] and — constructively — in Coq [5]. However, as far as we are aware the first serious formalization of arbitrary N-dimensional Euclidean space is quite recent, undertaken by Hales in HOL Light as part of the Flyspeck project:[2]

One reason for this may be that the basic real line suffices for many interesting applications such as the verification of floating-point algorithms. Another reason is that the proofs for N-dimensional space tend to be a bit technical, with a lot of summations and indexing, which makes them more tedious to formalize. A more substantial reason, however, may be that several theorem provers, including the numerous variants of HOL (HOL Light and Isabelle/HOL included) are based on a simple type theory where the type system seems to be more a hindrance than a help in formalizing N-ary Cartesian products. This applies not just to $\mathbb{R}^N$ but in other cases too — for example if one wants to formalize N-bit words as an N-ary Cartesian product $\texttt{bit}^N$ for some 2-element type `bit`. The problem with simple type theory is that a compound type can only depend

[1] The proofs and tools described are available within the '`Multiviariate`' directory of recent HOL Light releases available from `http://www.cl.cam.ac.uk/users/jrh/hol-light`.

[2] See `http://www.math.pitt.edu/~thales/flyspeck/index.html` for more about the project and the `Jordan` subdirectory of HOL Light for some of Hales's theories.

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 114–129, 2005.

on other *types* like $\mathbb{R}$ or `bit`, and not on *terms* like N. So how are N-ary Cartesian products A^N usually defined in HOL? We can identify two common approaches.

One can use a larger type such as (A)list or $\mathbb{N} \to A$ and identify subsets of it for particular N. While quite workable — this is what Hales's formalization does — it seems disappointing that the type system then makes little useful contribution, for example in 'automatically' ensuring that one does not take dot products of vectors of different lengths or wire together words of different sizes. All the interesting work is done by set constraints, just as if we were using an untyped system like set theory.

Alternatively, one can define specific instances for the N that are actually to be used; for example just $\texttt{bit}^{16}$ and $\texttt{bit}^{32}$ for a verification project or just $\mathbb{R}^2$ for a proof in plane geometry. However one may then need to duplicate structurally identical definitions, theorems and proofs for many different cases. One can use programmability to ease this burden; for example in HOL4 one can invoke an ML functor to create a specific word theory simply by:

```
structure word8Theory = wordFunctor (val bits = 8)
```

However, writing such a general theory requires a lot of rather tedious parametrization, and it seems inelegant to have various incompatible versions of what are naturally thought of as just instances of the same general result.

Of course, no problem arises in systems based on traditional set theory (such as Mizar) or those based on richer dependent type theories (such as Coq, Nuprl or PVS). So one might argue that it would be preferable to start from some such foundation rather than find ingenious ways to work around the deficiencies of a simpler one. However, simple type theory is a well-understood system with some appealing technical qualities such as efficiently decidable inference of most general types. Moreover, there has already been a considerable effort expended in developing comprehensive libraries of theorems and suites of proof tools for several provers based on simple type theory, and from a social point of view it would be difficult to abandon it.

A different approach to the problem of formalizing real vector spaces is to avoid Cartesian products by not using a 'basis' representation. For example, the IMPS system, which is also based on simple type theory, has been used to formalize analysis at a quite abstract level [3]. One can work in a theory of vector spaces over an arbitrary type '`V`', and then where necessary deduce the dimensions or choose a basis. However, one then needs a degree of overparametrization in all the results to indicate the ambient vector space operations and assert that they satisfy the required properties. By contrast, our solution below needs no such parametrization.

2 Our Formalization of $\mathbb{R}^N$

While HOL's version of simple type theory does not permit dependent types, it does feature polymorphic type variables. Our basic idea is to use types in place of numeric parameters, with the cardinality of the type being the dimension of the Cartesian product. That is, our formalization of A^N for a variable N is essentially the function space $N \to A$ where N is a *type* variable. In order to instantiate N to a particular value in

a theorem, we simply type-instantiate it so that the type replacing N has the appropriate size. For example, given a theorem about $\mathbb{R}^N$ with N a type variable, we can later specialize it to $\mathbb{R}^3$ by type-instantiating N to a 3-element type, which we can define as follows (there is no problem using the numeral 3 as the name of a type since types and terms belong to different namespaces):

```
let three_INDUCT,three_RECURSION =
  define_type "3 = three_1 | three_2 | three_3";;
```

One may object that, just as with the HOL4 word theory, one still needs to define a new type for each concrete instance required. However, this is now only an indexing type. All the definitions and theorems are generic, and one just needs to type-instantiate them to get specific instances. And in fact, one does not need to define new types for each instance. One can already create an N-element type by applying the sum-type constructor '$+$' to the one-element type '`1`':

$$\overbrace{1 + \cdots + 1}^{N \text{ times}}$$

This amounts to expressing the size of the indexing set in unary. More exotically, one can define type constructors for the construction of a binary or decimal representation [1].

Variants

Actually, there are at least three slightly different variants of the idea that we have considered:

- Literally use the function space $N \to \mathbb{R}$. This allows any indexing type, finite or infinite.
- Use a subset of functions $N \to \mathbb{R}$ with 'finite support', i.e. so that $\{x : N \mid f(x) \neq 0\}$ is finite.
- Use a subset of the set of functions $N \to \mathbb{R}$ that somehow 'forces N to be finite'.

These all have different strengths and weaknesses. The first alternative is the simplest and most transparent, but if certain theorems depend on N's finiteness we need to add an explicit hypothesis `FINITE(UNIV:N->bool)`. The second approach would probably be the most appropriate for a theory of vector spaces with finite or infinite dimension. But since we are mainly concerned with the finite-dimensional case, we have adopted the third alternative. More explicitly, we have defined a binary type constructor, written as infix '`^`', such that the type '`A^N`' is isomorphic to the usual function space $N \to A$ if N is finite and to A itself if N is infinite. In other words, we force infinite indexing sets to be treated as if they had size 1. This is accomplished by the following definition of a unary type constructor `finite_image`:

```
let finite_image_tybij =
  new_type_definition "finite_image" ("mk_finite_image","dest_finite_image")
   (prove(‘∃x:A. (x = @z. T) ∨ FINITE(UNIV:A->bool)‘,MESON_TAC[]));;
```

This ensures that the type `(A)finite_image` has the same size as `A` when that size is finite, and size `1` otherwise. The size of `(A)finite_image` can be determined by applying this function:

```
|- dimindex(s:A->bool) = if FINITE(UNIV:A->bool) then CARD(UNIV:A->bool) else 1
```

Now the binary type constructor `^` is defined so that `A^N` is isomorphic to the modified function space `(N)finite_image->A`, which yields the desired effect. These encoding tricks may be a bit obscure, but the end effect is that we can freely use the naive notation `A^N` while assuming whenever necessary that `N` is finite.

Since this is not actually the usual function space constructor, we need a corresponding notion of application and abstraction. Actually, the actual indexing type itself is of no interest, only its size. When we use it as for indexing, we would prefer, for conformance with informal convention, to use natural number indices from 1 to N. So we define an indexing operator, written as an infix `$` symbol, and with type $A^N \to \mathbb{N} \to A$, which internally picks some canonical way of mapping $\{1, \ldots, N\}$ bijectively into the indexing type and then applies it. We also define a corresponding notion of function abstraction (`lambda` written with binder syntax), and these satisfy the key property:

```
|- ∀i. 1 <= i ∧ i <= dimindex(UNIV:B->bool) ⇒ (((lambda) g:A^B) $i = g i)
```

For most purposes, one can now forget the coding details and use '`x$i`' where informally one would write x_i for indexing.

3 Vectors and Linear Algebra

It's now straightforward to define the basic operations on vectors. Addition and similar 'pointwise' operations are defined according to the following pattern:

```
|- x + y = lambda i. x$i + y$i
```

Note that we overload the usual arithmetic symbols like '$+$', but that the underlying constant on the left is actually `vector_add:real^N->real^N->real^N`, whereas the '$+$' on the right is the usual addition of real numbers. We define scalar multiplication of vectors by constants:

```
|- c % x = lambda i. c * x$i
```

and an injection from natural numbers, useful to denote the zero vector by `vec 0`:

```
|- vec n = lambda i. &n
```

More interesting is the inner ('dot') product. We show here the definition with the appropriate type annotations. Note that this looks quite close to the way this would be written informally, $x \cdot y = \sum_{i=1}^{n} x_i y_i$, except that since our `N` is a *type*, we need to convert it to a number by applying `dimindex` to its universe set:

```
|- (x:real^N) dot (y:real^N) =
     sum(1..dimindex(UNIV:N->bool)) (λi. x$i * y$i)
```

A Simple Decision Procedure

The basic algebraic properties of vectors can be derived fairly mechanically from the above definitions. In fact, we've hacked together a crude proof procedure that is able to prove most of the basic algebraic properties automatically by reducing them to the real case on the subcomponents. This is very convenient for generating the kinds of simple algebraic identities one often needs in proofs. Some of these (such as associativity of vector addition) are so useful that we bind them to names. More ad hoc lemmas can be generated dynamically.

```
# VECTOR_ARITH '∀x y:real^N. (x - y = vec 0) ⇔ (x = y)';;
val it : thm = |- ∀x y. (x - y = vec 0) ⇔ (x = y)
# VECTOR_ARITH '∀a b x:real^N. a % (b % x) = (a * b) % x';;
val it : thm = |- ∀a b x. a % b % x = (a * b) % x
# VECTOR_ARITH '∀x y z:real^N. (x + y) dot z = (x dot z) + (y dot z)';;
val it : thm = |- ∀x y z. (x + y) dot z = x dot z + y dot z
# VECTOR_ARITH '∀c x y:real^N. x dot (c % y) = c * (x dot y)';;
val it : thm = |- ∀c x y. x dot c % y = c * (x dot y)
```

The reduction process inside is, for many 'pointwise' theorems, a simple equivalence, e.g. $x + y = y + x$ to $\forall i.\ 1 \leq i \leq n \Rightarrow x_i + y_i = y_i + x_i$. In more general cases involving dot products and richer logical structure, the componentwise versions are proved anyway. For example, to prove $x \cdot y = \mathbf{0} \Rightarrow y \cdot x = \mathbf{0}$, it is first reduced to the equivalent $\sum_{i=1}^{n} x_i y_i = 0 \Rightarrow \sum_{i=1}^{n} y_i x_i = 0$ and that is deduced from the (a priori stronger) componentwise implications $\forall i.\ 1 \leq i \leq n \Rightarrow x_i y_i = 0 \Rightarrow y_i x_i = 0$, which are trivial. Note that assuming the postulated fact is true without regard to dimension, then it is in particular true for 1-dimensional space ($n = 1$ above), so the componentwise form is *not* in fact any stronger.

Norms

We next define the usual norm:

```
|- norm x = sqrt(x dot x)
```

and the corresponding distance function:

```
|- dist(x,y) = norm(x - y)
```

While apparently straightforward, this does raise a slight bootstrapping problem. Although the existing HOL analysis theory includes a large suite of theorems about square roots, our long-term goal is to subsume that theory in the present more general one. Therefore, we want to generate from scratch any results about square roots that we need. Before commencing analysis proper we prove the following lemma:

```
|- a <= b ∧ f(a) IN e1 ∧ f(b) IN e2 ∧
   (∀e x. a <= x ∧ x <= b ∧ &0 < e
          ⇒ ∃d. &0 < d ∧ ∀y. abs(y - x) < d ⇒ dist(f(y),f(x)) < e) ∧
   (∀y. y IN e1 ⇒ ∃e. &0 < e ∧ ∀y'. dist(y',y) < e ⇒ y' IN e1) ∧
   (∀y. y IN e2 ⇒ ∃e. &0 < e ∧ ∀y'. dist(y',y) < e ⇒ y' IN e2) ∧
   ¬(∃x. a <= x ∧ x <= b ∧ f(x) IN e1 ∧ f(x) IN e2)
   ⇒ ∃x. a <= x ∧ x <= b ∧ ¬(f(x) IN e1) ∧ ¬(f(x) IN e2)
```

This looks somewhat ugly and complicated, but it condenses to a more natural statement using concepts yet to be defined. It simply says that given a continuous mapping out of the real interval $[a, b]$ that maps a and b respectively into points of open sets e_1 and e_2 that have no common points in the image $f[a, b]$, there must be a point x in the interval such that $f(x)$ is contained in neither of those sets.

Later this is used to yield some standard theorems of analysis such as the fact that a convex set is connected. But in the short term, we use it to justify the existence of square roots, so we can proceed with our theory. It's now fairly easy to prove the usual norm properties such as the triangle law

```
|- ∀x y. norm(x + y) <= norm(x) + norm(y)
```

and the Cauchy-Schwarz inequality:

```
|- ∀x y. abs(x dot y) <= norm(x) * norm(y)
```

An arguably more elegant alternative used by Arthan in the development of analysis in the ProofPower version of HOL is to start the development based on the L_1 norm $||x|| = \sum_{i=1}^{n} |x_i|$ and develop analysis normally. Once this infrastructure is set up, properties of square roots are trivial, and it's then straightforward to show that all the basic topological properties are the same under the L_1 and usual norms and so map any earlier theorems across.

Linear Algebra

For us, linear algebra is only a tool for use in analytical results, and we have not developed it very comprehensively. We define a summation operator `vsum` over vectors, define orthogonality

```
|- orthogonal x y ⇔ (x dot y = &0)
```

and linearity of functions:

```
|- linear (f:real^M->real^N) ⇔
       (∀x y. f(x + y) = f(x) + f(y)) ∧
       (∀c x. f(c % x) = c % f(x))
```

We do not define a specific type of matrices, but represent $M \times N$ matrices using our Cartesian product twice. The usual arithmetic operations are then defined by by a further pointwise lifting, with '`**`' overloaded for matrix-matrix and matrix-vector multiplication. For example matrix-matrix multiplication is defined by:

```
|- (A:real^N^M) ** (B:real^P^N) =
       lambda i j. sum(1..dimindex(UNIV:N->bool)) (λk. A$i$k * B$k$j)
```

Note that to make the indexing correspond to the usual row-column convention, we needed to represent $M \times N$ matrices as $(\mathbb{R}^N)^M$, not $(\mathbb{R}^M)^N$. If this is not considered palatable, it would be strightforward to define a new type, say `(M,N)matrix` and an indexing function on pairs of numbers. But if we ignore such details, note how

nicely our typed formalization enforces the compatibility requirements in operations like matrix multiplication: one can only multiply an $M \times N$ matrix by a $N \times P$ one and the result is an $M \times P$ one.

The crucial theorems for our later work involve the correspondence between matrices and linear operators, with `matrix` mapping from a linear operator to the corresponding matrix:

```
|- ∀A:real^N^M. linear(λx. A ** x)

|- ∀f:real^M->real^N. linear f ⇒ ∀x. matrix f ** x = f(x)

|- ∀f g. linear f ∧ linear g ⇒ (matrix(g o f) = matrix g ** matrix f)
```

We have undertaken only a very rudimentary formalization of dimension, linear independence etc., just enough to reach one lemma that we need later on, that left and right invertibility coincide for $N \times N$ matrices.

```
|- ∀A:real^N^N A':real^N^N. (A ** A' = mat 1) ⇔ (A' ** A = mat 1)
```

It would however be a nice exercise in formalization to round out this theory with all the usual results of linear algebra, along the lines of Japser Stein's formalization in Coq.

4 A Decision Procedure

While the naive `VECTOR_ARITH` above is very useful, it is incapable of proving deeper facts about vectors. We spent some time looking for information on the decidability of theories of vector spaces. In contrast to the detailed catalogue of decidability and undecidability results that are known for groups, rings and fields, we were unable to find any such results. We therefore asked Robert Solovay about the subject. He was also unaware of any existing body of results, but within a few days had invented and described to us via email [16] a comprehensive set of quantifier elimination procedures for several variants of the first-order theory of real vector spaces. (Solovay is of the opinion that he is probably not the first to arrive at these results, and if any readers have seen such things before, the author would be very interested to know about them.)

Although the full quantifier elimination procedures are probably impractical, we thought it worthwhile to implement a cut-down version which, in principle, will successfully prove all theorems in the first-order language of real vector spaces where (i) all quantifiers over vectors are universal, and (ii) they are true in infinite-dimensional space. The reader will see shortly where these restrictions arise.

Initial Reduction

The first step in the procedure is to eliminate most vector operations, in fact all except dot products between pairs of variables.

First we eliminate the norm, which is already taken to be defined by $|x| = \sqrt{x \cdot x}$, by replacing any atomic formula $P(||x||)$ involving a norm by $\forall c.\ 0 \leq c \wedge x \cdot x = c^2 \Rightarrow P(c)$. In fact we optimize this reduction in common special cases, e.g. mapping

$||x|| < ||y||$ to $x \cdot x < y \cdot y$. We also write away the distance function `dist` using its definition.

Now note that any vector equality $x = y$ is equivalent to $x - y = 0$, which is in its turn equivalent to $|x - y| = 0$ and so to $(x - y) \cdot (x - y) = 0$. This allows us to eliminate vector equality. Actually we follow Solovay's original procedure in using $x \cdot x = y \cdot y \wedge x \cdot y = x \cdot x$ (the chain of implications between these equivalents is easy to establish).

Now we can distribute dot products through any composite terms and constants using various obvious rules. Note that these can be applied until the dot product is applied to pairs of variables only.

$$\begin{aligned}
\mathbf{0} \cdot x &= \mathbf{0} \\
x \cdot \mathbf{0} &= \mathbf{0} \\
(cx) \cdot y &= c(x \cdot y) \\
x \cdot (cy) &= c(x \cdot y) \\
-x \cdot y &= -(x \cdot y) \\
x \cdot -y &= -(x \cdot y) \\
(x + y) \cdot z &= x \cdot z + y \cdot z \\
x \cdot (y + z) &= x \cdot y + x \cdot z \\
(x - y) \cdot z &= x \cdot z - y \cdot z \\
x \cdot (y - z) &= x \cdot y - x \cdot z
\end{aligned}$$

These steps are easily packaged up as a HOL tactic `SOLOVAY_INIT_TAC` which reduce the initial goal. For example, here we set the Cauchy-Schwarz inequality as our goal:

```
# g ‘∀x y:real^N. x dot y <= norm x * norm y‘;;
val it : goalstack = 1 subgoal (1 total)

‘∀x y. x dot y <= norm x * norm y‘
```

and apply the tactic:

```
# e SOLOVAY_INIT_TAC;;
val it : goalstack = 1 subgoal (1 total)

‘&0 <= u1 ∧ (u1 pow 2 = y dot y)
 ⇒ &0 <= u2 ∧ (u2 pow 2 = x dot x)
 ⇒ x dot y <= u2 * u1‘
```

Elimination of Dot Products

Note that given a vector w and a list of vectors $v_1, \ldots, v_n$, we can express w as a linear combination of the v_i together with one more vector u that is orthogonal to all the v_i. This result (essentially the Gram-Schmidt process) is easy to prove by induction. Here is our HOL formalization using iterated operations over lists:

```
|- ∀w vs. ∃u as.
       ALL (orthogonal u) vs ∧ (LENGTH as = LENGTH vs) ∧
       (w = ITLIST2 (λa v s. a % v + s) as vs u)
```

This allows us to replace quantification over vectors $w, v_1, \ldots, v_n$ with a quantification over $u, v_1, \ldots, v_n$ where u is orthogonal to all the v_i:

```
|- (∀w:real^N. P w vs) =
        (∀as u. ALL (orthogonal u) vs ∧ (LENGTH as = LENGTH vs)
                ⇒ P (ITLIST2 (λa v s. a % v + s) as vs u) vs)
```

We can now expand out any dot products $w \cdot v_k$ into $\sum_{i=1}^{n} a_i(v_i \cdot v_k)$; note that $u \cdot v_k$ vanishes because of the orthogonality hypothesis. We can similarly expand out any instances of $w \cdot w$, and it is only here that we get a dot product involving u, namely $u \cdot u$.

Now note that in an infinite-dimensional space we can choose $u \cdot u$ arbitrarily (so long as it's nonnegative), because we can find a vector of any length that is orthogonal to a finite set of vectors. So for the formula $P[u \cdot u]$ to hold for all vectors u orthogonal to the v_i, it is necessary and sufficient that $P[c]$ should hold for all $c \geq 0$. In the general case, this is no longer an equivalence, because if the $v_1, \ldots, v_k$ already span the whole space we can only have $u = 0$. However, the implication is always valid in one direction, so we simply prove the more general goal that $\forall c.\ 0 \leq c \Rightarrow P[c]$. We have set up a generic HOL rule SOLOVAY_RULE which automatically generates a suitable general theorem for each number of vectors $v_1, \ldots, v_n$, e.g.

```
# SOLOVAY_RULE 2;;
val it : thm =
  |- (∀v0 v1 c.
         &0 <= c
         ⇒ (∀h h'.
                P v0 v1 (v0 dot (h % v0 + h' % v1))
                (v1 dot (h % v0 + h' % v1))
                ((h % v0 + h' % v1) dot (h % v0 + h' % v1) + c)))
     ⇒ (∀v0 v1 w. P v0 v1 (v0 dot w) (v1 dot w) (w dot w))
```

and then SOLOVAY_REDUCE_TAC normalizes dot products by symmetry then generates the right instance of the theorem and applies it. For our running example we get:

```
# e(REPEAT SOLOVAY_REDUCE_TAC);;
val it : goalstack = 1 subgoal (1 total)

'&0 <= c'
 ⇒ &0 <= c
    ⇒ (∀h. &0 <= u1 ∧ (u1 pow 2 = h * h * (&0 + c')) + c)
            ⇒ &0 <= u2 ∧ (u2 pow 2 = &0 + c')
               ⇒ h * (&0 + c') <= u2 * u1)'
```

We have successfully reduced the original assertion to an assertion over the reals that always implies it, and will still be true provided the original assertion was true over an infinite-dimensional vector space (or one with a sufficiently large dimension).

Solving the Real Problem

Of course, we still need to prove the resulting formula over the reals. Since it is purely universal, we opt to use an experimental HOL implementation of techniques based on

finding sum-of-squares decompositions using semidefinite programming [14]. This, using the CSDP semidefinite programming system, solves our goal quite easily:

```
# time e (CONV_TAC REAL_SOS);;
...
Iter: 33 Ap: 1.00e+00 Pobj:  1.5728640e+06 Ad: 8.34e-01 Dobj:  1.5728641e+06
Iter: 34 Ap: 1.00e+00 Pobj:  1.5728640e+06 Ad: 6.90e-01 Dobj:  1.5728640e+06
Iter: 35 Ap: 7.30e-01 Pobj:  1.5728640e+06 Ad: 8.49e-01 Dobj:  1.5728640e+06
Success: SDP solved
...
Trying rounding with limit 1
Translating proof certificate to HOL
CPU time (user): 4.44
val it : goalstack = No subgoals
```

Other Examples

In the following example, applying the reduction yields a rather complicated-looking formula. The result can still be proved automatically by REAL_SOS, but now it takes about a minute:

```
|- ∀a x y:real^N. (y - x) dot (a - y) > &0 ⇒ norm(y - a) < norm(x - a)
```

Although our present version of Solovay's procedure is limited to universal vector quantifiers, existential quantifiers over reals are quite acceptable, as in the following lemma we used in connection with some separating hyperplane proofs:

```
|- x dot y > &0 ⇒ ∃u. &0 < u ∧ ∀v. &0 < v ∧ v <= u ⇒ norm(v % y - x) < norm x
```

After reduction, we get the following formula. (This is the raw form; the inequality in the conclusion is amenable to some algebraic simplification.)

```
‘&0 <= c′ ∧ &0 <= c ∧ &0 < h * c′
 ⇒ (∃u. &0 < u ∧
        (∀v. &0 < v ∧ v <= u
             ⇒ v * (v * (h * h * c′ + c) - h * c′) - (v * h * c′ - c′) < c′))‘
```

Unsurprisingly, we still have an existential quantifier in the reduced formula. This means we cannot solve it using REAL_SOS, but we can pull out the "big gun", a general quantifier elimination procedure for the reals implemented in HOL by Sean McLaughlin [11] based on Hörmander's method [8,4,2]. This proves (the universal closure of) the above formula in about 15 seconds.

5 Topology and Analysis

The acid test of our approach to formalizing Euclidean space is whether it allows us to keep the formalization of more serious mathematical developments looking clean and elegant without introducing any significant difficulties. To this end, we will survey briefly some work we have undertaken in formalizing elementary topology and analysis. We will show quite a few statements of theorems, and we believe they generally look

fairly natural. The only potential difficulty we have identified is that since type variables are not quite such first-class objects as numbers, it is not trivial to formalize theorems that depend on induction over dimension. However, this pattern of reasoning has only come up in two theorems considered here. In one case, proving that a bounded closed set is (sequentially) compact, a workaround was necessary, but we will describe one that seemed quite simple and effective and could probably handle many similar situations. In the other, Brouwer's fixed point theorem, the induction takes place at the level of the underlying combinatorial lemma, and therefore the details of the formalization of Euclidean space make no difference.

We define the usual notions of topology in Euclidean space. We start with the slightly more general notion of one set being open in another, since this 'localized' notion is sometimes important:

```
|- s open_in u ⇔
       s SUBSET u ∧
       ∀x. x IN s ⇒ ∃e. &0 < e ∧
                        ∀x'. x' IN u ∧ dist(x',x) < e ⇒ x' IN s
```

and derive from it the 'global' version:

```
|- ∀s. open s ⇔ s open_in UNIV
```

Similarly we define `closed_in` and `closed`, open and closed balls:

```
|- ball(x,e) = { y | dist(x,y) < e}

|- cball(x,e) = { y | dist(x,y) <= e}
```

interior, closure, boundedness, limits, continuity, uniform continuity and convergence of sequences (of vectors). We then proceed to the usual properties such as completeness (every Cauchy sequence is convergent) connectedness, and compactness (every sequence has a convergent subsequence)

```
|- compact s ⇔
       ∀f:num->real^N.
           (∀n. f(n) IN s)
           ⇒ ∃l r. l IN s ∧ (∀m n:num. m < n ⇒ r(m) < r(n)) ∧
                     ((f o r) --> l) sequentially
```

and derive a fairly comprehensive set of the usual classics of analysis. For example, here is the Banach fixed point theorem:

```
|- ∀f s c. complete s ∧ ¬(s = {}) ∧
           &0 <= c ∧ c < &1 ∧
           (IMAGE f s) SUBSET s ∧
           (∀x y. x IN s ∧ y IN s ⇒ dist(f(x),f(y)) <= c * dist(x,y))
           ⇒ ∃!x:real^N. x IN s ∧ (f x = x)
```

and here is the Heine-Borel theorem:

```
|- compact s ⇔
          ∀f. (∀t. t IN f ⇒ open t) ∧ s SUBSET (UNIONS f)
              ⇒ ∃f'. f' SUBSET f ∧ FINITE f' ∧ s SUBSET (UNIONS f')
```

The proofs are all fairly well-known and routine. One more interesting case arises in the proof of the following:

```
|- compact s ⇔ bounded s ∧ closed s
```

The crucial argument is that a bounded closed N-dimensional 'interval' (or 'box') is compact. This proceeds by induction on dimension. While the proof is quite straightforward, the induction argument needs a little reformulation for our framework because we cannot really perform induction over a *type*. So we stay within one type $\mathbb{R}^N$ and consider the result for sequences in the various sets $S_k = \{s | \forall k \geq n.\ s_k = 0\}$, performing induction on k until we real the dimension of N. While not really difficult, it's slightly messy. Inductive arguments over dimension are perhaps the main weakness of our type-based formulation.

We also define the usual topological notion of homeomorphism and show that it preserves topological properties such as compactness:

```
|- ∀s t. s homeomorphic t ⇒ (compact s ⇔ compact t)
```

In fact, we have the more general results that compactness and connectedness are preserved under continuous images:

```
|- f continuous_on s ∧ compact s ⇒ compact(IMAGE f s)

|- f continuous_on s ∧ connected s ⇒ connected(IMAGE f s)
```

We define the convexity of a set: the line segment between any two points of the set lies entirely in the set.

```
|- convex s ⇔
      ∀x y u v. x IN s ∧ y IN s ∧ &0 <= u ∧ &0 <= v ∧ (u + v = &1)
                ⇒ (u % x + v % y) IN s
```

We also define a generic notion of 'hull', written as an infix so we can then consider '`convex hull s`', '`affine hull s`', '`conic hull s`' without duplication of basic lemmas. We even use '`closed hull s`' as the definition of '`closure`'.

```
|- P hull s = INTERS {t | P t ∧ s SUBSET t}
```

We prove many of the classic 'separation' theorems for convex sets, e.g. strict separation for a closed and a compact set:

```
|- ∀s t. convex s ∧ compact s ∧ ¬(s = {}) ∧
         convex t ∧ closed t ∧ DISJOINT s t
         ⇒ ∃a:real^N b. (∀x. x IN s ⇒ a dot x < b) ∧
                        (∀x. x IN t ⇒ a dot x > b)
```

One key result is that all convex compact sets with nonempty interior are homeomorphic:

```
|- convex s ∧ compact s ∧ ¬(interior s = {}) ∧
   convex t ∧ compact t ∧ ¬(interior t = {})
   ⇒ s homeomorphic t
```

Our next major theorem — certainly the hardest to formalize of those presented here — is Brouwer's Fixed Point Theorem. Using the above homeomorphism property, it is sufficient to prove it for a convenient special case, and we use the unit cube:

```
|- f continuous_on (interval [vec 0,vec 1]) ∧
   IMAGE f (interval [vec 0,vec 1]) SUBSET (interval [vec 0,vec 1])
   ⇒ ∃x. x IN interval[vec 0,vec 1] ∧ (f x = x)
```

One approach to this theorem is to develop some more extensive machinery from algebraic topology. Since that was not our primary interest, we were originally planning to formalize the fairly elementary proof based on Sperner's combinatorial lemma. However, this requires the formalization of the intuitively clear fact that we can subdivide a standard N-dimensional simplex into arbitrarily small simplices (e.g. by barycentric subdivision). Instead, we settled on a different approach due to Kuhn [10], where we need only the much simpler result that we can chop a cube into arbitrarily small cubes. Still, the proof of the combinatorial lemma underlying Kuhn's proof required a lot of work to formalize, possibly because of a poor choice of formalization. Still, once we get Brouwer's theorem it's easy to deduce the usual consequences such as the absence of a retraction from a closed ball onto its boundary:

```
|- ∀a:real^N e. &0 < e ⇒ ¬(frontier(cball(a,e)) retract_of cball(a,e))
```

We now define the usual notion of derivative for vector functions. Following Frechet, the derivative is defined to be the linear function that approximates the function close to a point. We are accustomed to thinking of the derivative of a function $\mathbb{R} \to \mathbb{R}$ as simply a real number, but in this framework we think of it as the linear function resulting from multiplication by that number:

```
|- (f has_derivative f') (at x) ⇔
        linear f' ∧
        ((λy. inv(norm(y - x)) % (f(y) - (f(x) + f'(y - x)))) --> vec 0)
        (at x)
```

The matrix corresponding to the derivative is the Jacobian (with respect to the usual basis):

```
|- jacobian f net = matrix(frechet_derivative f net)
```

All the usual results such as derivatives of sums are easy to prove:

```
|- (f has_derivative f') net ∧ (g has_derivative g') net
       ⇒ ((λx. f(x) + g(x)) has_derivative (λh. f'(h) + g'(h))) net
```

and the 'chain rule' is also reasonably straightforward:

```
|- (f has_derivative f') (at x) ∧
   (g has_derivative g') (at (f x))
   ⇒ ((g o f) has_derivative (g' o f')) (at x)
```

We also prove an important generalization of the usual mean value theorem for $\mathbb{R} \to \mathbb{R}$ functions.

```
|- convex s ∧ open s ∧
   (∀x. x IN s ⇒ (f has_derivative f'(x)) (at x)) ∧
   (∀x. x IN s ⇒ onorm(f'(x)) <= B)
   ⇒ ∀x y. x IN s ∧ y IN s ⇒ norm(f(x) - f(y)) <= B * norm(x - y)
```

where `onorm` is the 'operator norm' of a linear function:

```
|- onorm (f:real^M->real^N) = sup { norm(f x) | norm(x) = &1 }
```

The most interesting result in this area is the inverse function theorem. It is customary to state this for a continuously differentiable function, but if one simply wants differentiability of the inverse function, the usual hypotheses are much stronger than necessary — of the analysis books we have examined only Rudin [15] makes this explicit. We use the following sharper open mapping theorem as a lemma — we took the proof from Sussmann [17], who refers to it as 'well known', though we've never seen it anywhere else. Note that this result is for a general function $f : \mathbb{R}^M \to \mathbb{R}^N$ without the assumption that $M = N$.

```
|- open s ∧ f continuous_on s ∧
   x IN s ∧ (f has_derivative f') (at x) ∧ linear g' ∧ (f' o g' = I)
   ⇒ ∀t. t SUBSET s ∧ x IN interior(t)
          ⇒ f(x) IN interior(IMAGE f t)
```

However, the usual inverse function theorem does require the restricted type $f : \mathbb{R}^N \to \mathbb{R}^N$:

```
|- open s ∧ x IN s ∧ f continuous_on s ∧
   (∀x. x IN s ⇒ (g(f(x)) = x)) ∧
   (f has_derivative f') (at x) ∧ (f' o g' = I)
   ⇒ (g has_derivative g') (at (f(x)))
```

In order to deduce the existence of the local inverse function from the invertibility of the derivative, we do seem to need continuity of the derivative, but only at a point:

```
|- a IN s ∧ open s ∧ linear g' ∧ (g' o f'(a) = I) ∧
   (∀x. x IN s ⇒ (f has_derivative f'(x)) (at x)) ∧
   (∀e. &0 < e
        ⇒ ∃d. &0 < d ∧
                ∀x. dist(a,x) < d ⇒ onorm(λv. f'(x) v - f'(a) v) < e)
   ⇒ ∃t. a IN t ∧ open t ∧
          ∀x x'. x IN t ∧ x' IN t ∧ (f x' = f x) ⇒ (x' = x)
```

We have proved some results on generalized power series (of linear operators) and have made a start on a theory of integration, but this work is still quite fragmentary and we will not describe it in more detail here.

6 Future Work

Our two main priorities are (1) to develop a theory of integration that can then be used for the Flyspeck project, and (2) to link up the existing real analysis theory so that the present one cleanly subsumes and generalizes it. We also want to make a link to Hales's theories of Euclidean space. At the moment neither subsumes the other. For example,

Hales proves the highly non-trivial Jordan Curve Theorem as well as some other results in topology that we do not (e.g. equivalence of connectedness and path-connectedness). Although the underlying formalizations of Euclidean space are different, they are isomorphic and it should be easy enough to transfer results automatically.

Another interesting line of work (but with no particular applications in view) is to formalize complex differentiability or some appropriate generalization. Complex differentiability can be considered as differentiability of a $\mathbb{R}^2 \to \mathbb{R}^2$ function with a skew-symmetric Jacobian (i.e. where the partial derivatives satisfy the Cauchy-Riemann equations). We may also want to formalize traditional vector calculus and/or the theory of differential forms. Ideally, one would like to deduce Cauchy's theorem as a special case of a generalized Stokes theorem, but one needs to pay attention to the details of the integration theory to make this work.

Our existing treatment of topology is fixed in Euclidean space. While for the most part this is attractive because of the lack of parametrization, there are situations where we want to consider topologies on other sets such as the space of linear operators or continuous functions. Note, for example, that one hypothesis in the last theorem above is nothing but continuity in the space of linear functions, but we need to 'expand out' the definition because it does not come within our existing setup. Perhaps it would be more attractive to generalize `open_in` and `closed_in` to arbitrary toplogies, not simply other subsets of Euclidean space. The modifications required to do this are not very extensive.

Acknowledgements

The idea of formalizing multivariate calculus in this way arose at a seminar at New York University, and in particular in conversation with Sean McLaughlin and Tom Hales. I want to thank Clark Barrett for inviting me and to those mentioned and Jacob Schwartz for stimulating discussions. My debt to Robert Solovay for the decision procedure has already been made explicit, and he also explained some topological results to me. Thanks also to the referees for some helpful suggestions.

References

1. M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In N. Benton and A. Kennedy, editors, *BABEL'01: First workshop on multi-language infrastructure and interoperability*, 2001. Available online via `http://docs.msdnaa.net/ark_new/Webfiles/babel.htm`.
2. J. Bochnak, M. Coste, and M.-F. Roy. *Real Algebraic Geometry*, volume 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, 1998.
3. W. M. Farmer and F. J. Thayer. Two computer-supported proofs in metric space topology. *Notices of the American Mathematical Society*, 38:1133–1138, 1991.
4. L. Gårding. *Some Points of Analysis and Their History*, volume 11 of *University Lecture Series*. American Mathematical Society / Higher Education Press, 1997.
5. H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs, Proceedings of the International Workshop, TYPES 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 96–111. Springer-Verlag, 2001.

6. J. Harrison. Constructing the real numbers in HOL. In L. J. M. Claesen and M. J. C. Gordon, editors, *Proceedings of the IFIP TC10/WG10.2 International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume A-20 of *IFIP Transactions A: Computer Science and Technology*, pages 145–164, IMEC, Leuven, Belgium, 1992. North-Holland.
7. J. Harrison. Complex quantifier elimination in HOL. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, pages 159–174. Division of Informatics, University of Edinburgh, 2001. Published as Informatics Report Series EDI-INF-RR-0046. Available on the Web at `http://www.informatics.ed.ac.uk/publications/report/0046.html`.
8. L. Hörmander. *The Analysis of Linear Partial Differential Operators II*, volume 257 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, 1983.
9. L. S. v. B. Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System*. PhD thesis, Eindhoven University of Technology, 1977. Useful summary in [13], pp. 701–732.
10. H. W. Kuhn. Some combinatorial lemmas in topology. *IBM Journal of research and development*, 4:518–524, 1960. Available on the Web from `http://www.research.ibm.com/journal/rd/045/ibmrd0405K.pdf`.
11. S. McLaughlin and J. Harrison. A proof-producing decision procedure for real arithmetic. To appear in proceedings of the 20th International Conference on Automated Deduction, 2005.
12. R. Milewski. Fundamental theorem of algebra. *Journal of Formalized Mathematics*, 12, 2000. See `http://mizar.org/JFM/Vol12/polynom5.html`.
13. R. P. Nederpelt, J. H. Geuvers, and R. C. d. Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.
14. P. Parrilo. Semidefinite programming relaxations for semialgebraic problems. Available from the Web at `citeseer.nj.nec.com/parrilo01semidefinite.html`, 2001.
15. W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, 3rd edition, 1976.
16. R. Solovay. Elimination of quantifiers I, II, III. Email messages to John Harrison, 9, 19, 20 and 27 November 2004, 2004.
17. H. J. Sussmann. Multidifferential calculus: chain rule, open mapping and transversal intersection theorems. In W. W. Hager and P. M. Pardalos, editors, *Optimal control: theory, algorithms, and applications*, pages 436–487. Kluwer, 1998.

A Design Structure for Higher Order Quotients

Peter V. Homeier

U.S. Department of Defense
homeier@saul.cis.upenn.edu
http://www.cis.upenn.edu/~homeier

Abstract. The quotient operation is a standard feature of set theory, where a set is partitioned into subsets by an equivalence relation. We reinterpret this idea for higher order logic, where types are divided by an equivalence relation to create new types, called quotient types. We present a design to mechanically construct quotient types as new types in the logic, and to support the automatic lifting of constants and theorems about the original types to corresponding constants and theorems about the quotient types. This design exceeds the functionality of Harrison's package, creating quotients of multiple mutually recursive types simultaneously, and supporting the equivalence of aggregate types, such as lists and pairs. Most importantly, this design supports the creation of higher order quotients, which enable the automatic lifting of theorems with quantification over functions of any higher order.

1 Introduction

The quotient operation is a standard feature of mathematics, including set theory and abstract algebra. It provides a way to cleanly identify elements that previously were distinct. This simplifies the system by removing unneeded structure.

Traditionally, quotients [4] have found many applications. Classic examples are the construction of the integers from pairs of non-negative natural numbers, or the rationals from pairs of integers. In the lambda calculus [1], it is common to identify terms which are alpha-equivalent, that differ only by the choice of local names used by binding operators. Other examples include the construction of bags from lists by ignoring order, and sets from bags by ignoring duplicates.

The ubiquity of quotients has recommended their investigation within the field of mechanical theorem proving. The first to appear was Ton Kalker's 1989 package for HOL88 [11]. Isabelle/HOL [14] has mechanical support for the creation of higher order quotients by Oscar Slotosch [19], using partial equivalence relations represented as a type class, with equivalence relations as a subclass. That system provides a definitional framework for establishing quotient types, including higher order. Independently, Larry Paulson has shown a construction of first-order quotients in Isabelle without any use of the Hilbert choice operator [17]. PVS uses quotients to support theory interpretations [15]. MetaPRL has quotients in its foundations, as a type with a new equality [16]. Coq, based on the Calculus of Constructions [9], supports first order quotients [5] but has some

J. Hurd and T.F. Melham (Eds.): HM 2005, LNCS 3603, pp. 130–146, 2005.

difficulties with higher order [3]. These systems provide little automatic support. In particular, there is no automatic lifting of constants or theorems.

John Harrison has developed a package for the HOL theorem prover which supports first order quotients, including automation to define new quotient types and to lift to the quotient level both constants and theorems previously established [8]. This automatic lifting is key to practical support for quotients. A quotient of a group would be incomplete without also mapping the original group operation to a corresponding one for the quotient group. Similarly, theorems about the group which are independent of the equivalence relation should also be true of the quotient group. Mechanizing this lifting is vital for avoiding the repetition of proofs at the higher level which were already proved at the lower level. Such automation is not only practical, but mathematically incisive.

Despite the quality of Harrison's excellent package, it does have limitations. It can only lift one type at a time, and does not deal with aggregate types, such as lists or pairs involving types being lifted, which makes it difficult to lift a family of mutually nested recursive types. Most importantly, it is limited to lifting only first order theorems, where quantification is permitted over the type being lifted, but not over functions or predicates involving the type being lifted.

In this paper we describe a design for a new package for quotients [10] for the Higher Order Logic theorem prover that meets all these concerns. It provides a tool for lifting multiple types across multiple equivalence relations simultaneously. Aggregate equivalence relations are produced and used automatically. But most significantly, this package supports the automatic lifting of theorems that involve higher order functions, including quantification, of any finite order. This is possible through the use of *partial equivalence relations* [2,18], as a possibly non-reflexive variant of equivalence relations, enabling the creation of quotients of function types. The relationship between these partial equivalence relations and their associated abstraction and representation functions (mapping between the lower and higher types) is expressed in *quotient theorems*, which are central.

The precise definition in section 3 of the quotient relationship between the original and lifted types, and the proof in section 5.2 of that relationship's preservation for a function type, given existing quotients for the function's domain and range, are the heart of this paper, and are presented in full detail. These form the core theory that justifies the treatment of all higher order functions, including higher order universal, existential, and unique existential quantification.

The structure of this paper is as follows. Section 2 discusses equivalence relations and their extensions. Section 3 defines partial equivalence relations and quotient theorems. Section 4 shows the construction of a new quotient type in HOL. Section 5 explains the extension of quotients for aggregate and function types. Section 6 explores an alternative design that avoids use of the Axiom of Choice. Section 7 touches on highlights of the implementation and its required inputs. Section 8 exhibits an example. Finally, section 9 presents our conclusions.

We thank the TPHOLs referees for their cogent and constructive comments. We are grateful for the helpful comments and suggestions made by Rob Arthan, Randolph Johnson, Sylvan Pinsky, Yvonne V. Shashoua, and Konrad Slind,

and especially Michael Mislove for identifying partial equivalence relations, and William Schneeberger for the key idea in the proof of Theorem 19.

2 Equivalence Relations and Equivalence Theorems

Before considering quotients, we examine equivalence relations, on which such traditional quotients as those mentioned in the introduction have been based.

Let τ be any type. A binary relation R on τ can be represented in HOL as a curried function of type $\tau \to (\tau \to \texttt{bool})$. We will take advantage of the curried nature of R, where $R\ x\ y = (R\ x)\ y$.

An equivalence relation is a binary relation E satisfying

$$\begin{array}{lll} \textit{reflexivity:} & \forall x : \tau. & E\ x\ x \\ \textit{symmetry:} & \forall x\ y : \tau. & E\ x\ y\ \Rightarrow\ E\ y\ x \\ \textit{transitivity:} & \forall x\ y\ z : \tau. & E\ x\ y\ \wedge\ E\ y\ z\ \Rightarrow\ E\ x\ z \end{array}$$

These three properties are encompassed in the *equivalence property:*

$$\textit{equivalence:} \qquad \texttt{EQUIV}\ E \stackrel{\text{def}}{=} \forall x\ y : \tau.\ E\ x\ y\ \Leftrightarrow\ (E\ x = E\ y)$$

A theorem of the form $\vdash \texttt{EQUIV}\ E$ is called an *equivalence theorem* on type τ.

2.1 Equivalence Extension Theorems

Given an equivalence relation $E : \tau \to \tau \to \texttt{bool}$ on values of type τ, there is a natural extension of E to values of lists of τ. This is expressed as `LIST_REL` E, which forms an equivalence relation of type τ `list` $\to \tau$ `list` $\to$ `bool`. Similarly, equivalence relations on pairs, sums, and options may be formed from their constituent types' equivalence relations by the following operators.

Type	Operator	Type of operator
list	`LIST_REL :`	`('a -> 'a -> bool) -> 'a list -> 'a list -> bool`
pair	`### :`	`('a -> 'a -> bool) -> ('b -> 'b -> bool) -> 'a # 'b -> 'a # 'b -> bool`
sum	`+++ :`	`('a -> 'a -> bool) -> ('b -> 'b -> bool) -> 'a + 'b -> 'a + 'b -> bool`
option	`OPTION_REL :`	`('a -> 'a -> bool) -> 'a option -> 'a option -> bool`

These operators are easily defined in the expected way [10]. They are used to build an equivalence relation with a structure analogous to the type operator structure of the type of the elements compared by the relation.

Using these relation extension operators, the aggregate type operators `list`, `prod`, `sum`, and `option` have the following *equivalence extension theorems:*

$\texttt{LIST\_EQUIV}: \vdash \forall E.\ \texttt{EQUIV}\ E \Rightarrow \texttt{EQUIV}\,(\texttt{LIST\_REL}\ E)$
$\texttt{PAIR\_EQUIV}: \vdash \forall E_1\ E_2.\ \texttt{EQUIV}\ E_1 \Rightarrow \texttt{EQUIV}\ E_2 \Rightarrow \texttt{EQUIV}\,(E_1\ \texttt{\#\#\#}\ E_2)$
$\texttt{SUM\_EQUIV}: \vdash \forall E_1\ E_2.\ \texttt{EQUIV}\ E_1 \Rightarrow \texttt{EQUIV}\ E_2 \Rightarrow \texttt{EQUIV}\,(E_1\ \texttt{+++}\ E_2)$
$\texttt{OPTION\_EQUIV}: \vdash \forall E.\ \texttt{EQUIV}\ E \Rightarrow \texttt{EQUIV}\,(\texttt{OPTION\_REL}\ E)$

3 Partial Equivalence Relations and Quotient Theorems

In this section we introduce a new definition of the quotient relationship, based on *partial equivalence relations* (PERs), related to but different from equivalence relations. Every equivalence relation is a partial equivalence relation, but not every partial equivalence relation is an equivalence relation. An equivalence relation is reflexive, symmetric and transitive, while a partial equivalence relation is symmetric and transitive, but not necessarily reflexive on all of its domain.

Why use partial equivalence relations with a weaker reflexivity condition? The reason involves forming quotients of higher order types, that is, functions whose domains or ranges involve types being lifted. Unlike lists and pairs, the equivalence relations for the domain and range do not naturally extend to a useful equivalence relation for functions from the domain to the range.

The reason is that not all functions which are elements of the function type are *respectful* of the associated equivalence relations, as described in [10]. For example, given an equivalence relation $E : \tau \to \tau \to \texttt{bool}$, the set of functions from τ to τ may contain a function $f^?$ where for some x and y which are equivalent ($E\ x\ y$), the results of $f^?$ are not equivalent ($\neg(E\ (f^?\ x)\ (f^?\ y))$). Such disrespectful functions cannot be worked with; they do not correspond to any function at the abstract quotient level. Suppose instead that $f^?$ did lift. Let $\lceil\phi\rceil$ be the lifted version of ϕ. As $\lceil f^?\rceil$ is the lifted version of $f^?$, it should act just like $f^?$ on its argument, except that it should not consider the lower level details that E disregards. Thus $\forall u.\ \lceil f^?\rceil\lceil u\rceil = \lceil f^?\ u\rceil$. Then certainly $\forall u\ v.\ E\ u\ v \Leftrightarrow (\lceil u\rceil = \lceil v\rceil)$, and because $E\ x\ y$, we must have $\lceil x\rceil = \lceil y\rceil$. Applying $\lceil f^?\rceil$ to both sides, $\lceil f^?\rceil\lceil x\rceil = \lceil f^?\rceil\lceil y\rceil$. But this implies $\lceil f^?\ x\rceil = \lceil f^?\ y\rceil$, which means that $E\ (f^?\ x)\ (f^?\ y)$, which we have said is false, a contradiction. Therefore such disrespectful functions cannot be lifted, and we must exclude them. Using partial equivalence relations accomplishes this exclusion.

First, we say an element r *respects* R if and only if $R\ r\ r$.

Definition 1 (Quotient). *A relation R with abstraction function abs and representation function rep (between the representation, lower type τ and the abstract, quotient type ξ) is a quotient (notated $\langle R,abs,rep\rangle$) if and only if*

$$
\begin{array}{ll}
(1) & \forall a : \xi.\ abs\ (rep\ a) = a \\
(2) & \forall a : \xi.\ R\ (rep\ a)\ (rep\ a) \\
(3) & \forall r, s : \tau.\ R\ r\ s \Leftrightarrow R\ r\ r \wedge R\ s\ s \wedge (abs\ r = abs\ s)
\end{array}
$$

Property 1 states that *rep* is a right inverse of *abs*.

Property 2 states that the range of *rep* respects R.

Property 3 states that two elements of τ are related by R if and only if each element respects R and their abstractions are equal.

These three properties (1-3) describe the way the partial equivalence relation R works together with *abs* and *rep* to establish the correct quotient relationship between the lower type τ and the quotient type ξ. The precise definition of this quotient relationship is a central contribution of this work. This relationship is defined in the HOL logic as a new predicate:

```
QUOTIENT (R:'a -> 'a -> bool) (abs:'a -> 'b) (rep:'b -> 'a) ⇔
     (∀a. abs (rep a) = a) ∧
     (∀a. R (rep a) (rep a)) ∧
     (∀r s. R r s ⇔ R r r ∧ R s s ∧ (abs r = abs s))
```

The relationship that R with *abs* and *rep* is a quotient is expressed in HOL as

$$\vdash \texttt{QUOTIENT}\ R\ abs\ rep\ .$$

A theorem of this form is called a *quotient theorem*. The identity is $\vdash \langle \texttt{\$=}, \texttt{I}, \texttt{I} \rangle$.

These three properties support the inference of a quotient theorem for a function type, given quotient theorems for the domain and the range. This key inference is central and necessary to enable higher order quotients.

4 Quotient Types

The user may specify a quotient of a type τ by a relation R (written τ/R) by giving either a theorem that the relation is an equivalence relation, of the form

$$\vdash \forall x\ y.\ R\ x\ y\ \Leftrightarrow\ (R\ x = R\ y)\ , \tag{1}$$

or one that the relation is a nonempty partial equivalence relation, of the form

$$\vdash (\exists x.\ R\ x\ x)\ \wedge\ (\forall x\ y.\ R\ x\ y\ \Leftrightarrow\ R\ x\ x \wedge R\ y\ y \wedge (R\ x = R\ y))\ . \tag{2}$$

In this section we will develop the second, more difficult case. The first follows immediately. In the following, x, y, r, $s : \tau$, $c : \tau \to \texttt{bool}$, and $a : \tau/R$.

New types may be defined in HOL using the function `new_type_definition` [6, sections 18.2.2.3-5]. This function requires us to choose a representing type, and a predicate on that type denoting a subset that is nonempty.

Definition 2. *We define the new quotient type τ/R as isomorphic to the subset of the representing type $\tau \to \texttt{bool}$ by the predicate $P : (\tau \to \texttt{bool}) \to \texttt{bool}$, where $P\ c \stackrel{\text{def}}{=} \exists x.\ R\ x\ x \wedge (c = R\ x)$.*

P is nonempty because $P\ (R\ x)$ for the $x : \tau$ such that $R\ x\ x$ by (2). Let $\xi = \tau/R$. The HOL tool `define_new_type_bijections` [6] automatically defines a function $abs_c : (\tau \to \texttt{bool}) \to \xi$ and its right inverse $rep_c : \xi \to (\tau \to \texttt{bool})$ satisfying

Definition 3. (a) $\forall a : \xi.\ abs_c\ (rep_c\ a) = a$
(b) $\forall c : \tau \to \texttt{bool}.\ P\ c\ \Leftrightarrow\ rep_c\ (abs_c\ c) = c$

PER classes are subsets of τ (of type $\tau \to \texttt{bool}$) which satisfy P. Then abs_c and rep_c map between the quotient type ξ and PER classes (hence the "c").

Lemma 4 (rep_c maps to PER classes). $\forall a.\ P\ (rep_c\ a)$.

Proof: By Definition 3(a), $abs_c\ (rep_c\ a) = a$, so taking the rep_c of both sides, $rep_c\ (abs_c\ (rep_c\ a)) = rep_c\ a$. By Definition 3(b), $P\ (rep_c\ a)$. □

Lemma 5. $\forall r.\ R\ r\ r \Rightarrow (rep_c\ (abs_c\ (R\ r)) = R\ r)$.

Proof: Assume $R\ r\ r$; then $P\ (R\ r)$. By Definition 3(b), the goal follows.

Lemma 6 (abs_c is one-to-one on PER classes).
$\forall r\ s.\ R\ r\ r \Rightarrow R\ s\ s \Rightarrow (abs_c\ (R\ r) = abs_c\ (R\ s)\ \Leftrightarrow\ R\ r = R\ s)$.

Proof: Assume $R\ r\ r$ and $R\ s\ s$. The right-to-left implication of the biconditional is trivial. Assume $abs_c\ (R\ x) = abs_c\ (R\ y)$. Applying rep_c to both sides gives us $rep_c\ (abs_c\ (R\ x)) = rep_c\ (abs_c\ (R\ y))$. Then by Lemma 5 twice, $R\ x = R\ y$. □

The functions abs_c and rep_c map between PER classes of type $\tau \to \texttt{bool}$ and the quotient type ξ. Using these functions, we can define new functions abs and rep between the original type τ and the quotient type ξ as follows.

Definition 7 (Quotient abstraction and representation functions).

$$\begin{array}{lll} abs : \tau \to \xi & abs\ r \stackrel{\text{def}}{=} & abs_c\ (R\ r) \\ rep : \xi \to \tau & rep\ a \stackrel{\text{def}}{=} & \texttt{\$@}\ (rep_c\ a)\ \ (=\ \texttt{@}r.\ rep_c\ a\ r\ ) \end{array}$$

The `@` operator is a higher order version of Hilbert's choice operator ε [6,12]. It has type $(\alpha \to \texttt{bool}) \to \alpha$, and is usually used as a binder, where $\texttt{\$@}\ P = \texttt{@}x.\ P\ x$. (The `$` converts an operator to prefix syntax.) `@` satisfies the HOL axiom $\forall P\ x.\ P\ x \Rightarrow P\ (\texttt{\$@}\ P)$. Given any predicate P on a type, if any element of the type satisfies the predicate, then $\texttt{\$@}\ P$ returns an arbitrary element of that type which satisfies P. If no element of the type satisfies P, then $\texttt{\$@}\ P$ will return simply some arbitrary, unknown element of the type. Such definitions have been questioned by constructivist critics of the Axiom of Choice. An alternative design for quotients avoiding the Axiom of Choice is described in section 6.

Lemma 8. $\forall r.\ R\ r\ r \Rightarrow (R\ (\texttt{\$@}\ (R\ r)) = R\ r)$.

Proof: The axiom for the `@` operator is $\forall P\ x.\ P\ x \Rightarrow P\ (\texttt{\$@}\ P)$. Taking $P = R\ r$ and $x = r$, we have $R\ r\ r \Rightarrow R\ r\ (\texttt{\$@}\ R\ r)$. Assuming $R\ r\ r$, $R\ r\ (\texttt{\$@}\ (R\ r))$ follows. Then by (2), $R\ r\ (\texttt{\$@}\ (R\ r))$ implies the equality $R\ r = R\ (\texttt{\$@}\ (R\ r))$. □

Theorem 9. $\forall a.\ abs\ (rep\ a) = a$

Proof: By Lemma 4 and the definition of P, for each a there exists an r such that $R\,r\,r$ and $rep_c\ a = R\,r$. Then by Lemma 8, $R\ (\texttt{\$@}\ (R\,r)) = R\,r$. Now by Definition 7, $abs\ (rep\ a) = abs_c\ (R\ (\texttt{\$@}\ (rep_c\ a)))$, which simplifies by the above and Definition 3(a) to a. □

Theorem 10. $\forall a.\ R\ (rep\ a)\ (rep\ a)$.

Proof: As before, for each a there exists an r such that $R\ r\ r$ and $rep_c\ a = R\ r$.

$$\begin{array}{lll} R\ (rep\ a)\ (rep\ a) & \Leftrightarrow\ R\ (\$@\ (rep_c\ a))\ (\$@\ (rep_c\ a)) & \text{Definition 7} \\ & \Leftrightarrow\ R\ (\$@\ (R\ r))\ (\$@\ (R\ r)) & \text{selection of } r \\ & \Leftrightarrow\ R\ r\ (\$@\ (R\ r)) & \text{Lemma 8} \\ & \Leftrightarrow\ R\ (\$@\ (R\ r))\ r & \text{symmetry of } R \\ & \Leftrightarrow\ R\ r\ r\ \Leftrightarrow\ T & \text{Lemma 8, selection of } r \end{array}$$

□

Theorem 11. $\forall r\ s.\ \ R\ r\ s\ \Leftrightarrow\ R\ r\ r\ \wedge\ R\ s\ s\ \wedge\ (abs\ r = abs\ s)$

Proof:

$$\begin{array}{lll} R\ r\ s & \Leftrightarrow\ R\ r\ r\ \wedge\ R\ s\ s\ \wedge\ (R\ r = R\ s) & (2) \\ & \Leftrightarrow\ R\ r\ r\ \wedge\ R\ s\ s\ \wedge\ (abs_c\ (R\ r) = abs_c\ (R\ s)) & \text{Lemma 6} \\ & \Leftrightarrow\ R\ r\ r\ \wedge\ R\ s\ s\ \wedge\ (abs\ r = abs\ s) & \text{Definition 7} \end{array}$$

□

Theorem 12. $\langle R,\ abs,\ rep\rangle$.

Proof: By Theorems 9, 10, and 11, with Definition 1. □

5 Aggregate and Higher Order Quotient Theorems

Traditional quotients that lift τ to a set of τ also lift lists of τ to sets of lists of τ. These sets are isomorphic to lists, but *they are not lists*. In this design, when τ is lifted to ξ, then we lift lists of τ to lists of ξ. We preserve the type operator structure built on top of the types being lifted. Similarly, we want to preserve polymorphic constants. In a theorem being lifted, we want an occurrence of $\texttt{HD} : \tau\ \texttt{list} \to \tau$ to lift to an occurrence of $\texttt{HD} : \xi\ \texttt{list} \to \xi$. If such a constant is not lifted to itself, the lifted version of the theorem will not look like the original. Hence Definition 1 was designed to preserve the vital type operator structure.

In the process of lifting constants and theorems, quotient theorems are needed for each argument and result type of each constant being lifted. For aggregate and higher order types, the tool automatically proves any needed quotient theorems from the available quotient theorems for the constituent subtypes. To accomplish this, the tool uses *quotient extension theorems* (section 5.2). These are provided preproven for some standard type operators. For others, new quotient extension theorems may be manually proven and then included to extend the tool's power.

5.1 Aggregate and Higher Order PERs and Map Operators

Some aggregate equivalence relation operators have been already described in section 2, and these can equally be used to build aggregate partial equivalence relations. In addition, for function types, the following is added:

Type Operator	Type of operator
fun `===>` :	`('a -> 'a -> bool) -> ('b -> 'b -> bool) -> ('a -> 'b) -> ('a -> 'b) -> bool`

Definition 13. $(R_1$ `===>` $R_2)\ f\ g \Leftrightarrow \forall x\ y.\ R_1\ x\ y \Rightarrow R_2\ (f\ x)\ (g\ y)$.

Note R_1 `===>` R_2 is *not* in general an equivalence relation (it is not reflexive). It is reflexive at a function f, $(R_1$ `===>` $R_2)\ f\ f$, if and only if f is respectful.

The quotient theorems created for aggregate types involve not only aggregate partial equivalence relations, but also aggregate abstraction and representation functions. These are constructed from the component abstraction and representation functions using the following "map" operators.

Type	Operator	Type of operator, examples of *abs* and *rep* fns
list	`MAP` :	`('a -> 'b) -> 'a list -> 'b list` *examples:* (`MAP` *abs*) , (`MAP` *rep*)
pair	`##` :	`('a -> 'b) -> ('c -> 'd) -> 'a # 'c -> 'b # 'd` *examples:* (abs_1 `##` abs_2) , (rep_1 `##` rep_2)
sum	`++` :	`('a -> 'b) -> ('c -> 'd) -> 'a + 'c -> 'b + 'd` *examples:* (abs_1 `++` abs_2) , (rep_1 `++` rep_2)
option	`OPTION_MAP` :	`('a -> 'b) -> 'a option -> 'b option` *examples:* (`OPTION_MAP` *abs*) , (`OPTION_MAP` *rep*)
fun	`-->` :	`('a -> 'b) -> ('c -> 'd) -> ('b -> 'c) -> 'a -> 'd` *examples:* (rep_1 `-->` abs_2) , (abs_1 `-->` rep_2)

The above operators are easily defined in the expected way [10], if not already present in standard HOL. The identity quotient map operator is the identity operator `I` : $\alpha \to \alpha$. The function map operator definition is of special interest:

Definition 14. $(f$ `-->` $g)\ h\ x \stackrel{\text{def}}{=} g\ (h\ (f\ x))$.

5.2 Quotient Extension Theorems

Here are the quotient extension theorems for the `list`, `prod`, `sum`, `option`, and, most significantly, `fun` type operators:

LIST_QUOTIENT:
$\vdash \forall R\ abs\ rep.\ \langle R, abs, rep\rangle \Rightarrow \langle \texttt{LIST\_REL}\ R,\ \texttt{MAP}\ abs,\ \texttt{MAP}\ rep\rangle$

PAIR_QUOTIENT:
$\vdash \forall R_1\ abs_1\ rep_1.\ \langle R_1, abs_1, rep_1\rangle \Rightarrow\ \forall R_2\ abs_2\ rep_2.\ \langle R_2, abs_2, rep_2\rangle \Rightarrow$
$\langle R_1\ \texttt{\#\#\#}\ R_2,\ \ abs_1\ \texttt{\#\#}\ abs_2,\ \ rep_1\ \texttt{\#\#}\ rep_2\rangle$

SUM_QUOTIENT:
$\vdash \forall R_1\ abs_1\ rep_1.\ \langle R_1, abs_1, rep_1\rangle \Rightarrow\ \forall R_2\ abs_2\ rep_2.\ \langle R_2, abs_2, rep_2\rangle \Rightarrow$
$\langle R_1\ \texttt{+++}\ R_2,\ \ abs_1\ \texttt{++}\ abs_2,\ \ rep_1\ \texttt{++}\ rep_2\rangle$

OPTION_QUOTIENT:
$\vdash \forall R\ abs\ rep.\ \langle R, abs, rep\rangle \Rightarrow$
$\langle \texttt{OPTION\_REL}\ R,\ \texttt{OPTION\_MAP}\ abs,\ \texttt{OPTION\_MAP}\ rep\rangle$

FUN_QUOTIENT:
$\vdash \forall R_1\ abs_1\ rep_1.\ \langle R_1, abs_1, rep_1\rangle \Rightarrow\ \forall R_2\ abs_2\ rep_2.\ \langle R_2, abs_2, rep_2\rangle \Rightarrow$
$\langle R_1\ \texttt{===>}\ R_2,\ \ rep_1\ \texttt{-->}\ abs_2,\ \ abs_1\ \texttt{-->}\ rep_2\rangle$

This last theorem is of central and critical importance to forming higher order quotients. We present here its proof in detail.

Theorem 15 (Function quotients). *If relations R_1 and R_2 with abstraction functions abs_1 and abs_2 and representation functions rep_1 and rep_2, respectively, are quotients, then R_1 ===> R_2 with abstraction function rep_1 --> abs_2 and representation function abs_1 --> rep_2 is a quotient.*

Proof: We need to prove the three properties of Definition 1:

Property 1. Prove for all a, $(rep_1\ \texttt{-->}\ abs_2)\ ((abs_1\ \texttt{-->}\ rep_2)\ a) = a$.
Proof: The equality here is between functions, and by extension, true if for all values x in a's domain, $(rep_1\ \texttt{-->}\ abs_2)\ ((abs_1\ \texttt{-->}\ rep_2)\ a)\ x = a\ x$.
By the definition of -->, this is $abs_2\ ((abs_1\ \texttt{-->}\ rep_2)\ a\ (rep_1\ x)) = a\ x$, and then $abs_2\ (rep_2\ (a\ (abs_1\ (rep_1\ x)))) = a\ x$. By Property 1 of $\langle R_1,abs_1,rep_1\rangle$, $abs_1\ (rep_1\ x) = x$, and by Property 1 of $\langle R_2,abs_2,rep_2\rangle$, $\forall b.\ abs_2\ (rep_2\ b) = b$, so this reduces to $a\ x = a\ x$, true.

Property 2. Prove $(R_1\ \texttt{===>}\ R_2)\ ((abs_1\ \texttt{-->}\ rep_2)\ a)\ ((abs_1\ \texttt{-->}\ rep_2)\ a)$.
Proof: By the definition of ===>, this is
$\forall x, y.\ R_1\ x\ y\ \Rightarrow R_2\ ((abs_1\ \texttt{-->}\ rep_2)\ a\ x)\ ((abs_1\ \texttt{-->}\ rep_2)\ a\ y)$. Assume $R_1\ x\ y$, and show $R_2\ ((abs_1\ \texttt{-->}\ rep_2)\ a\ x)\ ((abs_1\ \texttt{-->}\ rep_2)\ a\ y)$. By the definition of -->, this is $R_2\ (rep_2\ (a\ (abs_1\ x)))\ (rep_2\ (a\ (abs_1\ y)))$. Now since $R_1\ x\ y$, by Property 3 of $\langle R_1,abs_1,rep_1\rangle$, $abs_1\ x = abs_1\ y$. Substituting this into our goal, we must prove $R_2\ (rep_2\ (a\ (abs_1\ y)))\ (rep_2\ (a\ (abs_1\ y)))$. But this is an instance of Property 2 of $\langle R_2,abs_2,rep_2\rangle$, and so the goal is proven.

Property 3. Prove $(R_1\ \texttt{===>}\ R_2)\ r\ s \Leftrightarrow$
$(R_1\ \texttt{===>}\ R_2)\ r\ r \wedge (R_1\ \texttt{===>}\ R_2)\ s\ s \wedge ((rep_1\ \texttt{-->}\ abs_2)\ r = (rep_1\ \texttt{-->}\ abs_2)\ s)$.

The last conjunct on the right side is equality between functions, so by extension this is $(R_1 \texttt{===>} R_2)\ r\ s \Leftrightarrow (R_1 \texttt{===>} R_2)\ r\ r \wedge (R_1 \texttt{===>} R_2)\ s\ s \wedge$
$(\forall x.\ (rep_1 \texttt{-->}\ abs_2)\ r\ x = (rep_1 \texttt{-->}\ abs_2)\ s\ x)$.
By the definitions of ===> and -->, this is (1) $\Leftrightarrow$ (2) $\wedge$ (3) $\wedge$ (4), where

$$\begin{array}{ll} (1) & (\forall x\ y.\ R_1\ x\ y\ \Rightarrow R_2\ (r\ x)\ (s\ y)) \\ (2) & (\forall x\ y.\ R_1\ x\ y\ \Rightarrow R_2\ (r\ x)\ (r\ y)) \\ (3) & (\forall x\ y.\ R_1\ x\ y\ \Rightarrow R_2\ (s\ x)\ (s\ y)) \\ (4) & (\forall x.\ (abs_2\ (r\ (rep_1\ x)) = abs_2\ (s\ (rep_1\ x))). \end{array}$$

We prove (1) $\Leftrightarrow$ (2) $\wedge$ (3) $\wedge$ (4) as a biconditional with two goals.

Goal 1. ($\Rightarrow$) Assume (1). Then we must prove (2), (3), and (4).

Subgoal 1.1. (Proof of (2)) Assume $R_1\ x\ y$. We must prove $R_2\ (r\ x)\ (r\ y)$. From $R_1\ x\ y$ and Property 3 of $\langle R_1,abs_1,rep_1\rangle$, we also have $R_1\ x\ x$ and $R_1\ y\ y$. From (1) and $R_1\ x\ y$, we have $R_2\ (r\ x)\ (s\ y)$. From (1) and $R_1\ y\ y$, we have $R_2\ (r\ y)\ (s\ y)$. Then by symmetry and transitivity of R_2, the goal is proven.

Subgoal 1.2. (Proof of (3)) Similar to the previous subgoal.

Subgoal 1.3. (Proof of (4)) $R_1\ (rep_1\ x)\ (rep_1\ x)$ follows from Property 2 of $\langle R_1,abs_1,rep_1\rangle$. From (1), we have $R_2\ (r\ (rep_1\ x))\ (s\ (rep_1\ x))$. Then the goal follows from this and the third conjunct of Property 3 of $\langle R_2,abs_2,rep_2\rangle$.

Goal 2. ($\Leftarrow$) Assume (2), (3), and (4). We must prove (1). Assume $R_1\ x\ y$. Then we must prove $R_2\ (r\ x)\ (s\ y)$. From $R_1\ x\ y$ and Property 3 of $\langle R_1,abs_1,rep_1\rangle$, we also have $R_1\ x\ x$, $R_1\ y\ y$, and $abs_1\ x = abs_1\ y$. By Property 3 of $\langle R_2,abs_2,rep_2\rangle$, the goal is $R_2\ (r\ x)\ (r\ x) \wedge R_2\ (s\ y)\ (s\ y) \wedge abs_2\ (r\ x) = abs_2\ (s\ y)$. This breaks into three subgoals.

Subgoal 2.1. Prove $R_2\ (r\ x)\ (r\ x)$. This follows from $R_1\ x\ x$ and (2).

Subgoal 2.2. Prove $R_2\ (s\ y)\ (s\ y)$. This follows from $R_1\ y\ y$ and (3).

Subgoal 2.3. Prove $abs_2\ (r\ x) = abs_2\ (s\ y)$.

> **Lemma.** *If $\langle R,abs,rep\rangle$ and $R\ z\ z$, then $R\ (rep\ (abs\ z))\ z$.*
> $R\ (rep\ (abs\ z))\ (rep\ (abs\ z))$, by Property 2 of $\langle R,abs,rep\rangle$. From the hypothesis, $R\ z\ z$. From Property 1 of $\langle R,abs,rep\rangle$, $abs\ (rep\ (abs\ z)) = abs\ z$. From these three statements and Property 3 of $\langle R,abs,rep\rangle$, we have $R\ (rep\ (abs\ z))\ z$. □

By the lemma and $R_1\ x\ x$, we have $R_1\ (rep_1\ (abs_1\ x))\ x$. Similarly, by the lemma and $R_1\ y\ y$, we have $R_1\ (rep_1\ (abs_1\ y))\ y$. Then by (2), we have $R_2\ (r\ (rep_1\ (abs_1\ x)))\ (r\ x)$, and by (3), $R_2\ (s\ (rep_1\ (abs_1\ y)))\ (s\ y)$. From these and Property 3 of $\langle R_2,abs_2,rep_2\rangle$,

$$\begin{array}{l} abs_2\ (r\ (rep_1\ (abs_1\ x))) = abs_2\ (r\ x) \text{ and} \\ abs_2\ (s\ (rep_1\ (abs_1\ y))) = abs_2\ (s\ y). \end{array}$$

But by $abs_1\ x = abs_1\ y$ and (4), the left hand sides of these two equations are equal, so their right hand sides must be also, $abs_2\ (r\ x) = abs_2\ (s\ y)$, which proves the goal. □

6 The Axiom of Choice

Gregory Moore wrote that "Rarely have the practitioners of mathematics, a discipline known for the certainty of its conclusions, differed so vehemently over one of its central premises as they have done over the Axiom of Choice. Yet without the Axiom, mathematics today would be quite different" [13]. Today, this discussion continues. Some theorem provers are based on classical logic, and others on a constructivist logic. In higher order logic, the Axiom of Choice is represented by Hilbert's ε-operator [12, §4.4], also called the indefinite description operator. Paulson's lucid recent work [17] exhibits an approach to quotients which avoids the use of Hilbert's ε-operator, by instead using the definite description operator ι [14, §5.10]. These two operators may be axiomatized as follows:

$$\begin{array}{rcl} \forall P\ x.\ P\ x \Rightarrow P(\varepsilon\ P) & \quad or \quad & \forall P.\ (\exists x.\ P\ x) \Rightarrow P(\varepsilon\ P) \\ \forall P\ x.\ P\ x \Rightarrow (\forall y.\ P\ y \Rightarrow x = y) \Rightarrow P(\iota\ P) & \quad or \quad & \forall P.\ (\exists! x.\ P\ x) \Rightarrow P(\iota\ P) \end{array}$$

The ι-operator yields the single element of a singleton set, $\iota\{z\} = z$, but its result on non-singleton sets is indeterminate. By contrast, the ε-operator chooses some indeterminate element of any non-empty set, even if nondenumerable. The ι-operator is weaker than the ε-operator, and less objectionable to constructivists.

Thus it is of interest to determine if a design for higher order quotients may be formulated using only ι, not ε. Inspired by Paulson, we investigate this by forming an alternative design, eliminating the representation functions.

Definition 16 (Constructive quotient, replacing Definition 1). *A relation R with abstraction function abs (between the representation type τ and the abstraction type ξ) is a quotient (notated as $\langle R, abs\rangle$) if and only if*

$$\begin{array}{ll} (1) & \forall a : \xi.\ \exists r : \tau.\ R\ r\ r \wedge (abs\ r = a) \\ (2) & \forall r\ s : \tau.\ R\ r\ s \Leftrightarrow R\ r\ r \wedge R\ s\ s \wedge (abs\ r = abs\ s) \end{array}$$

Property 1 states that for every abstract element a of ξ there exists a representative in τ which respects R and whose abstraction is a.

Property 2 states that two elements of τ are related by R if and only if each element respects R and their abstractions are equal.

The quotients for new quotient types based on (partial) equivalence relations may now be constructed by a modified version of §4, where the representation function *rep* is omitted from Definition 7, so there is no use of the Hilbert ε-operator. Property 1 follows from Lemma 4. The identity quotient is $\langle \texttt{\$=}, \texttt{I}\rangle$. From Definition 16 also follow analogs of the quotient extension theorems, e.g.,

$$\forall R\ abs.\ \langle R,\ abs\rangle \Rightarrow \langle \texttt{LIST\_REL}\ R,\ \texttt{MAP}\ abs\rangle$$

for lists and similarly for pairs, sums and option types. Functions are lifted by the abstraction operation for functions, which requires two new definitions:

$$\begin{array}{rcl} (abs \Downarrow R)\ a\ r & \stackrel{\text{def}}{=} & R\ r\ r\ \wedge\ abs\ r = a \\ (reps\ \texttt{+->}\ abs)\ f\ x & \stackrel{\text{def}}{=} & \iota\ (\texttt{IMAGE}\ abs\ (\texttt{IMAGE}\ f\ (reps\ x))) \end{array}$$

Note that for the identity quotient, $(\texttt{I} \Downarrow \texttt{\$=}) = \texttt{\$=}$.

The critical quotient extension theorem for functions has also been proven:

Theorem 17 (Function quotient extension).

$$\langle R_1,\ abs_1\rangle \Rightarrow \langle R_2,\ abs_2\rangle \Rightarrow \langle R_1\ \texttt{===>}\ R_2,\ (abs_1 \Downarrow R_1)\ \texttt{+->}\ abs_2\rangle$$

Unfortunately, the proof requires using the Axiom of Choice. In fact, this theorem implies the Axiom of Choice, in that it implies the existence of an operator which obeys the axiom of the Hilbert ε-operator, as seen by the following development.

Theorem 18 (Partial abstraction quotients). *If f is any function from type α to β, and Q is any predicate on α, such that $\forall y{:}\beta.\ \exists x{:}\alpha.\ Q\ x \wedge (f\ x = y)$, then the partial equivalence relation $R = \lambda r\ s.\ Q\ r \wedge Q\ s \wedge (f\ r = f\ s)$ with abstraction function f is a quotient ($\langle R, f\rangle$).*

Proof: Follows easily from substituting R in Definition 16 and simplifying. □

Theorem 19 (Partial inverses exist). *If f is any function from type α to β, and Q is any predicate on α, such that $\forall y{:}\beta.\ \exists x{:}\alpha.\ Q\ x \wedge (f\ x = y)$, then there exists a function g such that $f \circ g = \texttt{I}$ and $\forall y.\ Q\ (g\ y)$.* [William Schneeburger]

Proof: Assuming the function quotient extension theorem 17, we apply it to two quotient theorems; first, the identity quotient $\langle \texttt{\$=}, \texttt{I}\rangle$ for type β, and second, the partial abstraction quotient $\langle R, f\rangle$ from Theorem 18. This yields the quotient $\langle \texttt{\$=}\ \texttt{===>}\ R,\ \texttt{\$=}\ \texttt{+->}\ f\rangle$, since $(\texttt{I} \Downarrow \texttt{\$=}) = \texttt{\$=}$. By Property 1 of Definition 16, $\forall a.\ \exists r.\ (\texttt{\$=}\ \texttt{===>}\ R)\ r\ r \wedge ((\texttt{\$=}\ \texttt{+->}\ f)r = a)$. Specializing $a = \texttt{I} : \beta \to \beta$, and renaming r as g, we obtain $\exists g.\ (\texttt{\$=}\ \texttt{===>}\ R)\ g\ g \wedge (\texttt{\$=}\ \texttt{+->}\ f)g = \texttt{I})$. By the definition of ===>, $(\texttt{\$=}\ \texttt{===>}\ R)g\ g$ is $\forall x\ y.\ x = y \Rightarrow R\ (g\ x)\ (g\ y)$, which simplifies by the definition of R to $\forall y.\ Q\ (g\ y)$. The right conjunct is $(\texttt{\$=}\ \texttt{+->}\ f)g = \texttt{I}$, which by the definition of +-> is $(\lambda x.\ \iota\ (\texttt{IMAGE}\ f\ (\texttt{IMAGE}\ g\ (\texttt{\$=}\ x)))) = \texttt{I}$. But $\texttt{\$=}\ x$ is the singleton $\{x\}$, so since $\texttt{IMAGE}\ h\ \{z\} = \{h\ z\}$, $\iota\{z\} = z$, and $(\lambda x.\ f\ (g\ x)) = f \circ g$, this simplifies to $f \circ g = \texttt{I}$, and the conclusion follows. □

Theorem 20 (Existence of Hilbert choice). *There exists an operator $c : (\alpha \to \texttt{bool}) \to \alpha$ which obeys the Hilbert choice axiom, $\forall P\ x.\ P\ x \Rightarrow P\ (c\ P)$.*

Proof: In Theorem 19, let $Q = (\lambda(P{:}\alpha \to \texttt{bool},\ a{:}\alpha).\ (\exists x.\ P\ x) \Rightarrow P\ a)$ and $f = \texttt{FST}$. Then its antecedent is $\forall P'.\ \exists (P, a).\ ((\exists x.P\ x) \Rightarrow P\ a) \wedge (\texttt{FST}(P, a) = P')$. For any P', take $P = P'$, and if $\exists x.\ P\ x$, then take a to be such an x. Otherwise take a to be any value of α. In either case the antecedent is true. Therefore by Theorem 19 there exists a function g such that $\texttt{FST} \circ g = \texttt{I}$ and $\forall P.\ Q\ (g\ P)$, which is $\forall P.\ (\exists x.\ (\texttt{FST}\ (g\ P))\ x) \Rightarrow (\texttt{FST}\ (g\ P))\ (\texttt{SND}\ (g\ P))$. The operator c is taken as $\texttt{SND} \circ g$, and since $\texttt{FST}\ (g\ P) = P$, the Hilbert choice axiom follows. □

The significance of Theorem 20 is that even if we are able to avoid all use of the Axiom of Choice up to this point, it is not possible to prove the function quotient extension theorem 17 without it. This section's design may be used to build a theory of quotients which is constructive and which extends naturally to

quotients of lists, pairs, sums, and options. However, it is not possible to extend it to higher order quotients while remaining constructive. Therefore the designs presented in this paper cannot be used to create higher order quotients in strictly constructive theorem provers. Alternatively, in theorem provers like HOL which admit the Hilbert choice operator, if higher order quotients are desired, there is no advantage in avoiding using the Axiom of Choice through using the design of this section. The main design presented earlier is much simpler to automate.

7 Implementation

The design for higher order quotients presented here has been implemented in a package for the Higher Order Logic theorem prover. This section will touch on only a few interesting aspects of the implementation; for further details, see [10].

This implementation provides a tool which accomplishes all the three tasks of lifting types, constants, and theorems. To do these, the tool requires inputs of several kinds. For each new quotient type to be created, the user must provide a (partial) equivalence theorem (§4). For each kind of aggregate type involved, the user must provide a quotient extension theorem, and if possible, an equivalence extension theorem. For every constant which is mentioned in a theorem to be lifted, there must be a *respectfulness theorem* showing that the constant respects the equivalence relations. In addition, for polymorphic constants that can apply to arguments of either the lower or the quotient types, both a respectfulness theorem and a *preservation theorem* must be provided, which shows that the function of the polymorphic constant is preserved across the quotient operation.

$$\texttt{COND\_RSP:}\ \langle R, abs, rep\rangle \Rightarrow (a_1 = a_2)\ \wedge\ R\ b_1\ b_2\ \wedge\ R\ c_1\ c_2\ \Rightarrow R\ (\textbf{if}\ a_1\ \textbf{then}\ b_1\ \textbf{else}\ c_1)\ (\textbf{if}\ a_2\ \textbf{then}\ b_2\ \textbf{else}\ c_2)$$

$$\texttt{COND\_PRS:}\ \langle R, abs, rep\rangle \Rightarrow \textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c = abs\ (\textbf{if}\ a\ \textbf{then}\ rep\ b\ \textbf{else}\ rep\ c)$$

$$\texttt{RES\_FORALL\_RSP:}\ \langle R, abs, rep\rangle \Rightarrow (R\ \texttt{===>}\ \texttt{\$=})\ f\ g\ \Rightarrow \texttt{RES\_FORALL(respects}\ R)\ f = \texttt{RES\_FORALL(respects}\ R)\ g$$

$$\texttt{FORALL\_PRS:}\ \langle R, abs, rep\rangle \Rightarrow (\texttt{\$}\forall\ f = \texttt{RES\_FORALL(respects}\ R)\ ((abs\ \texttt{-->}\ \texttt{I})\ f)$$

Interestingly, $\forall$ is not respectful. To lift, theorems using $\forall$ are automatically converted to ones using `RES_FORALL`. `RES_FORALL` (`respects` R) P is the universal quantification of P, restricted to values of the argument of P which respect R. A large number of these respectfulness and preservation theorems have been preproven for standard operators, including, e.g., the unique existential quantifier. The natural power of higher order quotients is smoothly exploited in enabling these respectfulness and preservation theorems to be used to lift theorems containing curried operators with none, some, or all of their arguments present.

8 Example: Finite Sets

To demonstrate the higher order quotients package, we create finite sets as a new type, starting from the existing type of lists, `'a list`.

Lists are represented in HOL as a free algebra with two distinct constructors, `NIL` and `CONS`, also written as `[]` and infix `::` respectively. Let A, B, C be lists.

We intend to create the new type of finite sets as the quotient of lists by the equivalence relation $\sim$, generated by rule induction on the following six rules:

$$\frac{}{a::(b::A) \sim b::(a::A)} \qquad \frac{}{[] \sim []} \qquad \frac{A \sim B}{B \sim A}$$

$$\frac{}{a::(a::A) \sim a::A} \qquad \frac{A \sim B}{a::A \sim a::B} \qquad \frac{A \sim B, \quad B \sim C}{A \sim C}$$

It is easy to prove that $\sim$ is in fact an equivalence relation, reflexive, symmetric, and transitive, and so `fset_EQUIV:` $\vdash \forall A\ B.\ A \sim B \Leftrightarrow (\$\sim A = \$\sim B)$.

Theorems may be proved by induction using the list induction principle:

$$\forall P : \texttt{'a list} \to \texttt{bool}.\ P\ [] \ \wedge\ (\forall t.\ P\ t \Rightarrow \forall h.\ P\ (h::t)) \ \Rightarrow\ \forall l.\ P\ l$$

Membership and concatenation (which lifts to "union") are predefined:

$$\begin{array}{rcl@{\quad\wedge\quad}rcl} \texttt{MEM}\ x\ [] & = & F & \texttt{MEM}\ x\ (h::t) & = & (x = h) \vee \texttt{MEM}\ x\ t \\ \texttt{APPEND}\ []\ l & = & l & \texttt{APPEND}\ (h::l_1)\ l_2 & = & h::(\texttt{APPEND}\ l_1\ l_2) \end{array}$$

We define new constants by primitive recursion, and prove extensionality:

$$\begin{array}{rcl} [] \ \texttt{Delete1}\ x & = & [] \\ (a::A)\ \texttt{Delete1}\ x & = & \textbf{if}\ a = x\ \textbf{then}\ A\ \texttt{Delete1}\ x\ \textbf{else}\ a::(A\ \texttt{Delete1}\ x) \end{array}$$

$$\begin{array}{rcl} \texttt{Fold1}\ f\ g\ z\ [] & = & z \\ \texttt{Fold1}\ f\ g\ z\ (a::A) & = & \textbf{if}\ (\forall u\ v.\ f\ u\ v = f\ v\ u) \wedge \\ & & \quad (\forall u\ v\ w.\ f\ u\ (f\ v\ w) = f\ (f\ u\ v)\ w) \\ & & \textbf{then if}\ \texttt{MEM}\ a\ A\ \textbf{then}\ \texttt{Fold1}\ f\ g\ z\ A \\ & & \qquad\qquad \textbf{else}\ f\ (g\ a)\ (\texttt{Fold1}\ f\ g\ z\ A) \\ & & \textbf{else}\ z \end{array}$$

$$A \sim B \ \Leftrightarrow\ \forall a.\ \texttt{MEM}\ a\ A \Leftrightarrow \texttt{MEM}\ a\ B$$

Before invoking the quotient package, we must first prove the respectfulness theorems of each of the operators we wish to lift, `NIL_RSP`, `CONS_RSP`, etc., e.g.,

$$\frac{}{[] \sim []} \qquad \frac{A \sim B}{a::A \sim a::B} \qquad \frac{A \sim B}{\texttt{MEM}\ a\ A = \texttt{MEM}\ a\ B} \qquad \frac{A_1 \sim A_2, \quad B_1 \sim B_2}{\texttt{APPEND}\ A_1\ B_1 \sim \texttt{APPEND}\ A_2\ B_2}$$

$$\frac{A \sim B}{\texttt{Card1}\ A = \texttt{Card1}\ B} \qquad \frac{A \sim B}{A\ \texttt{Delete1}\ a \sim B\ \texttt{Delete1}\ a} \qquad \frac{A \sim B}{\texttt{Fold1}\ f\ g\ z\ A = \texttt{Fold1}\ f\ g\ z\ B}$$

We intend to lift the following constants on lists to new ones on finite sets:

`[]` $\mapsto$ `Empty`	`MEM` $\mapsto$ `In`	`APPEND` $\mapsto$ `Union`	`Delete1` $\mapsto$ `Delete`
`::` $\mapsto$ `Insert`	`Card1` $\mapsto$ `Card`	`Inter1` $\mapsto$ `Inter`	`Fold1` $\mapsto$ `Fold`

We now create the new type `'a finite_set` from the quotient of lists by $\sim$.

```
val [In, Union, finite_set_EXTENSION, ... finite_set_INDUCT] =
 define_quotient_types{
  types = [{name = "finite_set", equiv = fset_EQUIV}],
  defs=[{def_name="In_def",fname="In", fixity=Infix(NONASSOC,425),
         func=``MEM:'a -> 'a list -> bool``}, ... ],
  tyop_equivs = [],
  tyop_quotients = [FUN_QUOTIENT],
  tyop_simps = [FUN_REL_EQ, FUN_MAP_I],
  respects = [NIL_RSP, CONS_RSP, MEM_RSP, APPEND_RSP,
              Card1_RSP, Delete1_RSP, Inter1_RSP, Fold1_RSP],
  poly_preserves = [FORALL_PRS, EXISTS_PRS, COND_PRS],
  poly_respects = [RES_FORALL_RSP, RES_EXISTS_RSP, COND_RSP],
  old_thms = [MEM, APPEND, list_EXTENSION, ... list_INDUCT]};
```

This proves and stores the quotient theorem

$$\vdash \texttt{QUOTIENT \$}\sim \texttt{ finite\_set\_ABS finite\_set\_REP}.$$

It also defines the lifted versions of the constants, for example

$$\vdash \forall T_1\ T_2.\ T_1\ \texttt{Insert}\ T_2 = \texttt{finite\_set\_ABS}\ (T_1\ \texttt{::}\ \texttt{finite\_set\_REP}\ T_2)$$

The theorems listed in `old_thms` are automatically soundly lifted to the quotient level, with the types changed, now concerning not lists but finite sets, e.g.,

$$\begin{array}{rcl@{\quad}rcl}
x\ \texttt{In Empty} & = & F & x\ \texttt{In}\ (h\ \texttt{Insert}\ t) & = & (x = h) \vee x\ \texttt{In}\ t \\
\texttt{Empty Union}\ l & = & l & (h\ \texttt{Insert}\ l_1)\ \texttt{Union}\ l_2 & = & h\ \texttt{Insert}\ (l_1\ \texttt{Union}\ l_2)
\end{array}$$

$$\begin{array}{rcl}
\texttt{Empty Delete}\ x & = & \texttt{Empty} \\
(a\ \texttt{Insert}\ A)\ \texttt{Delete}\ x & = & \textbf{if}\ a = x\ \textbf{then}\ A\ \texttt{Delete}\ x \\
 & & \qquad\qquad \textbf{else}\ a\ \texttt{Insert}\ (A\ \texttt{Delete}\ x) \\
\texttt{Fold}\ f\ g\ z\ \texttt{Empty} & = & z \\
\texttt{Fold}\ f\ g\ z\ (a\ \texttt{Insert}\ A) & = & \textbf{if}\ (\forall u\ v.\ f\ u\ v = f\ v\ u) \wedge \\
 & & \quad (\forall u\ v\ w.\ f\ u\ (f\ v\ w) = f\ (f\ u\ v)\ w) \\
 & & \textbf{then if}\ a\ \texttt{In}\ A\ \textbf{then}\ \texttt{Fold}\ f\ g\ z\ A \\
 & & \qquad\qquad \textbf{else}\ f\ (g\ a)\ (\texttt{Fold}\ f\ g\ z\ A) \\
 & & \textbf{else}\ z
\end{array}$$

$$A = B\ \Leftrightarrow\ \forall a.\ a\ \texttt{In}\ A \Leftrightarrow a\ \texttt{In}\ B$$

$$\forall P.\ P\ \texttt{Empty}\ \wedge\ (\forall t.\ P\ t \Rightarrow \forall h.\ P\ (h\ \texttt{Insert}\ t))\ \Rightarrow\ \forall l.\ P\ l$$

The **if** ... **then** ... **else** in the `Delete` definition now yields a finite set, not a list. The last theorem requires higher order quotients to lift, because it involves quantification over functions, in this case P of type `'a finite_set` $\rightarrow$ `bool`.

9 Conclusions

We have presented a design for mechanically creating higher order quotients which is a conservative, definitional extension of higher order logic. The package

implemented from this design [10] automatically lifts not only types, but also constants and theorems from the original level to the quotient level.

The relationship between the lower type and the quotient type is characterized by the partial equivalence relation, the abstraction function, and the representation function. As a key contribution, three necessary algebraic properties have been identified for these to properly describe a quotient, which are preserved in the creation of both aggregate and higher order quotients.

The Axiom of Choice was used in this design. We showed that an alternative design may be constructed without dependence on the Axiom of Choice, but that it may not be extended to higher order quotients while remaining constructive.

Prior to this work, only Harrison [8] went beyond support for modeling the quotient types to provide automation for the lifting of constant definitions and theorems from their original statements concerning the original types to the corresponding analogous statements concerning the new quotient types. This is important for the practical application of quotients to sizable problems like quotients on the syntax of complex, realistic programming or specification languages. These may be modelled as recursive types, where terms which are partially equivalent by being well-typed and alpha-equivalent are identified by taking quotients. This eases the traditional problem of the capture of bound variables [7].

Such quotients may now be more easily and practically modeled within a variety of theorem provers, using the design described here.

Soli Deo Gloria.

References

1. Barendregt, H.P.: *The Lambda Calculus, Syntax and Semantics.* North-Holland, 1981.
2. Bruce, K., Mitchell, J. C.: 'PER models of subtyping, recursive types and higher-order polymorphism', in *Principles of Programming Languages 19*, Albequerque, New Mexico, 1992, pp. 316-327.
3. Chicli, L., Pottier, L., Simpson, C.: 'Mathematical Quotients and Quotient Types in Coq', Proceedings of *TYPES 2002*, Lecture Notes in Computer Science, vol. 2646 (Springer-Verlag, 2002).
4. Enderton, H. B.: *Elements of Set Theory.* Academic Press, 1977.
5. Geuvers, H., Pollack, R., Wiekijk, F., Zwanenburg, J.: 'A constructive algebraic hierarchy in Coq', in *Journal of Symbolic Computation*, 34(4), 2002, pp. 271-286.
6. Gordon, M. J. C., Melham, T. F.: *Introduction to HOL.* Cambridge University Press, Cambridge, 1993.
7. Gordon, A. D., Melham, T. F.: 'Five Axioms of Alpha Conversion', in *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, edited by J. von Wright, J. Grundy and J. Harrison, Lecture Notes in Computer Science, vol. 1125 (Springer-Verlag, 1996), pp. 173-190.
8. Harrison, J.: *Theorem Proving with the Real Numbers,* §2.11, pp. 33-37. Springer-Verlag 1998.
9. Hofmann, M.: 'A simple model for quotient types,' in *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science, vol. 902 (Springer-Verlag, 1995), pp. 216-234.

10. Homeier, P. V.: 'Higher Order Quotients in Higher Order Logic.' In preparation; draft available at `http://www.cis.upenn.edu/~hol/quotients`.
11. Kalker, T.: at `www.ftp.cl.cam.ac.uk/ftp/hvg/info-hol-archive/00xx/0082`.
12. Leisenring, A. C.: *Mathematical Logic and Hilbert's ε-Symbol.* Gordon and Breach, 1969.
13. Moore, G. H.: *Zermelo's Axiom of Choice: It's Origins, Development, and Influence.* Springer-Verlag 1982.
14. Nipkow, T., Paulson, L. C., Wenzel, M.: *Isabelle/HOL.* Springer-Verlag 2002.
15. Owre, S., Shankar, N.: *Theory Interpretations in PVS*, Technical Report SRI-CSL-01-01, Computer Science Lab., SRI International, Menlo Park, CA, April 2001.
16. Nogin, A.: 'Quotient Types: A Modular Approach,' in *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, edited by V. A. Carreño, C. Muñoz, and S. Tahar, Lecture Notes in Computer Science, vol. 2410 (Springer-Verlag, 2002), pp. 263-280.
17. Paulson, L.: 'Defining Functions on Equivalence Classes,' *ACM Transactions on Computational Logic*, in press. Previously issued as Report, Computer Lab, University of Cambridge, April 20, 2004.
18. Robinson, E.: 'How Complete is PER?', in *Fourth Annual Symposium on Logic in Computer Science* (LICS), 1989, pp. 106-111.
19. Slotosch, O.: 'Higher Order Quotients and their Implementation in Isabelle HOL', in *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, edited by Elsa L. Gunter and Amy Felty, Lecture Notes in Computer Science, vol. 1275 (Springer-Verlag, 1997), pp. 291-306.

Axiomatic Constructor Classes in Isabelle/HOLCF

Brian Huffman, John Matthews, and Peter White

OGI School of Science and Engineering at OHSU, Beaverton, OR 97006
{brianh, johnm, peterw}@cse.ogi.edu

Abstract. We have definitionally extended Isabelle/HOLCF to support axiomatic Haskell-style constructor classes. We have subsequently defined the functor and monad classes, together with their laws, and implemented state and resumption monad transformers as generic constructor class instances. This is a step towards our goal of giving modular denotational semantics for concurrent lazy functional programming languages, such as GHC Haskell.

1 Introduction

The Isabelle generic theorem prover is organized as a modular collection of tools for reasoning about a variety of logics. This allows common theorem proving tasks such as parsing, pretty printing, simplification of formulas, and proof search tactics to be reused in each object logic. We are similarly interested in using Isabelle to modularly reason about programs written in a wide spectrum of programming languages. In particular, we want to verify programs written in lazy functional programming languages, such as Haskell [11].

Denotational semantics is one attractive approach for this, owing to its high level of abstraction and the ease in which both recursive datatypes and functions can be modeled. However, language features change over time. Modeling new language datatypes and primitives usually only requires local changes to the language semantics. On the other hand, the introduction of a new computational effect, such as exceptions, often requires global modifications. Furthermore, these effects must be formalized anew for each programming language under consideration.

In the last decade *monads* have become an increasingly popular way to mitigate this problem. A monad M is a single-argument type constructor together with a particular set of operations and equational laws (listed in Section 4.4) that can be used for expressing kinds of computational effects, such as imperative state update, exception handling, and concurrency. For more information, we recommend consulting one of the Haskell-oriented monad tutorials at [`http://www.haskell.org/bookshelf/#monads`].

When formalizing the semantics of a language, a monad can be defined that models the language's computational effects. The rest of the semantics is then specified abstractly in terms of the monad, allowing the language's effects to be modified in isolation.

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 147–162, 2005.

Even greater modularity can be achieved by composing complex monads through a series of *monad transformers*. A monad transformer takes an existing monad and extends it with a specific new computational effect, such as imperative state, or exception handling. In this way an effect can be specified once, as a monad tranformer, and then reused in other language semantics.

1.1 Axiomatic Constructor Classes

Isabelle supports overloaded constant definitions [21]. Polymorphic constants usually have a single definition that covers all type instances, but multiple definitions are allowed if they apply to separate types. For example, unary negation in Isabelle/HOL has the polymorphic type $'a \Rightarrow 'a$. It may be applied to sets or to integers, with a different meaning in each case.

Axiomatic type classes [21] are another important feature of Isabelle. Each class has a set of *class axioms*, each of which has a single free type variable, and specifies properties of overloaded constants. An axiomatic type class is then a set of types: those types for which the class axioms have been proven to hold. Theorems can express assumptions about types using class constraints on type variables. Axiomatic type classes are used extensively in Isabelle's implementation of domain theory (see Section 2.1).

Unfortunately, Isabelle's type class system is not powerful enough to specify classes for monads or monad transformers. The problem is that Isabelle supports abstraction over types, using type variables; but there is no such abstraction for type constructors—they can only be used explicitly. This means that we can prove the monad laws hold for a particular type constructor, but we can not reason abstractly about monads in general. Furthermore, monad transformers cannot even be defined, since Isabelle does not allow type constructors to take other type constructors as arguments.

A key observation is that we can represent types themselves as values, and we can represent continuous type constructors as continuous functions over those values. We can then use Isabelle's existing type definition and axiomatic type class packages to represent such type constructors as new types. This now allows us to use type variables to reason abstractly about type constructors, and thus we can specify constructor classes simply as type classes.

This representation is carried out definitionally, and we have gone on to encode several monads and monad transformers in Isabelle/HOLCF. The most interesting of these is the resumption monad transformer, which models interleaving of computations.

2 Background

Isabelle is a generic interactive theorem prover, which can be instantiated with various kinds of object-logics. Isabelle/HOL is an instantiation of higher order logic. We will now summarize the basic syntax and keywords of Isabelle/HOL that will be used in the paper.

The formula syntax in Isabelle/HOL includes standard logical notation for connectives and quantifiers. In addition, Isabelle has separate syntax for the meta-level logic: $\bigwedge$, $\Longrightarrow$, and $\equiv$ represent meta-level universal quantification, implication, and equality. There is also notation for nested meta-level implication: $[\![P_1; \ldots; P_n]\!] \Longrightarrow R$ is short for $P_1 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow R$.

The syntax of types is similar to the language ML, except that Isabelle uses a double arrow ($\Rightarrow$) for function types. Some binary type constructors are written infix, as in the product type $nat \times bool$; other type constructors are written postfix, as in *bool list* or *nat set*. Finally, $'a$ and $'b$ denote free type variables.

Isabelle theories declare new constants with the **consts** keyword. Definitions may be supplied later using **defs**; alternatively, constants may be declared and defined at once using **constdefs**. Theories introduce new types with the **typedef** command, which defines a type isomorphic to a given non-empty set. The keywords **lemma** and **theorem** introduce theorems.

2.1 Isabelle/HOLCF

HOLCF [14,20] is an object logic for Isabelle designed for reasoning about functional programs. It is implemented as a layer on top of Isabelle/HOL, so it includes all the theories and syntax of the HOL object logic. In addition, HOLCF defines a family of new axiomatic type classes, several new type constructors, and associated syntax, which we will summarize here.

HOLCF introduces an overloaded binary relation $\sqsubseteq$, which is used to define information orderings for types: The proposition $x \sqsubseteq y$ means that x is an approximation to y. HOLCF then defines a sequence of axiomatic type classes $po \supseteq cpo \supseteq pcpo$ to assert properties of the $\sqsubseteq$ relation. The class *po* contains types where $\sqsubseteq$ defines a partial order. The subclass *cpo* is for ω-complete partial orders, which means that there exists a least upper bound for each ω-chain. An ω-chain Y is a countable sequence where $Y_n \sqsubseteq Y_{n+1}$ for all n. The expression $\bigsqcup_n Y_n$ denotes the least upper bound of the chain Y. Finally, the class *pcpo* is for ω-cpos that additionally have a least element, written $\bot$. HOLCF declares *pcpo* to be the default class, so free type variables are assumed to be in class *pcpo* unless otherwise specified.

HOLCF defines a standard set of type constructors from domain theory. Given types $'a$ and $'b$ in class *pcpo*, and $'c$ in class *cpo*, the following are all in class *pcpo*: the cartesian product $'a \times 'b$, the strict product $'a \otimes 'b$, the strict sum $'a \oplus 'b$, the lifted type $'a\ u$, and the continuous function space $'c \to 'a$. Recall that a continuous function is a monotone function that preserves limits: $f(\bigsqcup_n Y_n) = \bigsqcup_n f(Y_n)$. HOLCF also defines a type constructor *lift* that can turn any type into a flat pcpo by adding a new bottom element.

HOLCF defines special syntax for operations involving the continuous function space. Continuous function application is written with an infix dot, as in $f{\cdot}x$. Continuous lambda abstraction is written $\Lambda x.\ P$. Composition of continuous functions f and g is written $f\ oo\ g$.

3 Representing Types and Type Constructors

3.1 Embedding-Projection Pairs

To do formal reasoning about types, we need to be able to talk about what it means to embed one type into another. An appropriate concept in domain theory is the embedding-projection pair, or ep-pair [1,2,7]. Let $'a$ and $'b$ be types in class *pcpo*. A pair of continuous functions $e::'a \to 'b$ and $p::'b \to 'a$ is an ep-pair if $p \mathbin{oo} e = ID::'a \to 'a$ and $e \mathbin{oo} p \sqsubseteq ID::'b \to 'b$. The existence of such a pair shows that the type $'a$ can be embedded into type $'b$. An illustration of a simple ep-pair is shown in Fig. 1.

constdefs
is-ep-pair :: $('a \to 'b) \Rightarrow ('b \to 'a) \Rightarrow bool$
is-ep-pair $e\ p \equiv (\forall x::'a.\ p{\cdot}(e{\cdot}x) = x) \wedge (\forall y::'b.\ e{\cdot}(p{\cdot}y) \sqsubseteq y)$

Ep-pairs have many useful properties: e is injective, p is surjective, both are strict, each function uniquely determines the other, and the range of e is a sub-pcpo of $'b$. Ep-pairs are also compositional, and they can be lifted over many type constructors, including cartesian product and continuous function space.

If we identify $'a$ with a subset of $'b$, so that e is just subset inclusion, then it is natural to consider just the composition $e \mathbin{oo} p::'b \to 'b$. This gives a continuous function that is below the identity, and also idempotent, since $e \mathbin{oo} p \mathbin{oo} e \mathbin{oo} p = e \mathbin{oo} ID \mathbin{oo} p = e \mathbin{oo} p$. In domain theory, a function with these properties is called a projection (not to be confused with the second half of an ep-pair). Our definitions of ep-pairs and projections follow Amadio and Curien's [2, Defn. 7.1.6].

constdefs
is-projection :: $('a \to 'a) \Rightarrow bool$
is-projection $p \equiv (\forall x.\ p{\cdot}(p{\cdot}x) = p{\cdot}x) \wedge (\forall x.\ p{\cdot}x \sqsubseteq x)$

A projection is a function, but it can also be viewed as a set: Just take the range of the function, or equivalently, its set of fixed points—for idempotent

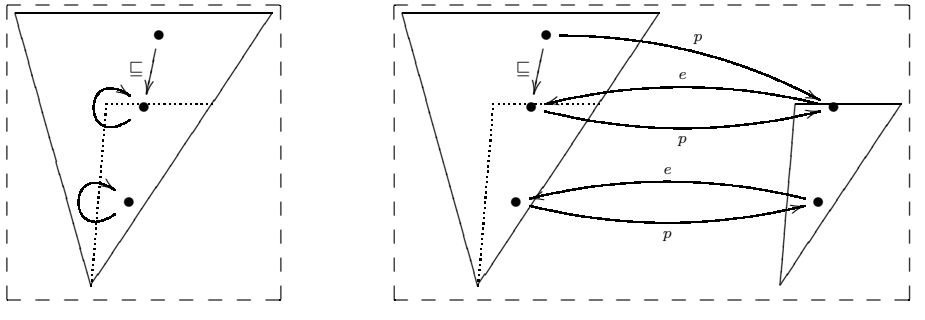


Fig. 1. Projections (*left*) and ep-pairs (*right*). The range of a projection defines a subset of a pcpo. An ep-pair defines an isomorphism between a subset and another type.

functions they are the same. A projection along with the set it defines are shown in Fig. 1. Every projection gives a set that is a sub-pcpo, and contains $\bot$. Not all sub-pcpos have a corresponding projection, but if one exists then it is unique. The set-oriented and function-oriented views of projections even give the same ordering: For any projections p and q, $p \sqsubseteq q$ if and only if *range* $p \subseteq$ *range* q.

We define a type constructor in Isabelle for the space of projections over any pcpo. Since *is-projection* is an admissible predicate, the set of projections is closed with respect to limits of ω-chains. Since $\Lambda x.\ \bot$ is a projection, the set also contains a least element—thus the resulting type is a pcpo.

typedef *'a projection* = {*p*::*'a* → *'a*. *is-projection p*}

Isabelle's type definition package provides *Rep* and *Abs* functions to convert between type *'a projection* and type *'a* → *'a*. *Rep-projection* is injective and its range equals the set of all projection functions; *Abs-projection* is a left-inverse to *Rep-projection*. (See the Isabelle Tutorial [16, §8.5.2] for more details.)

We can define some operations on projections that are useful for reasoning about projections as sets. The *in-projection* relation is like the $\in$ relation for sets; the triple-colon notation is meant to be reminiscent of type annotations in Isabelle or Haskell. The function *cast* is implemented by simply applying a projection as a function—in the set-oriented view of projections, the intuition is that it casts values into a set, preserving those values that are already in the set.

constdefs
cast :: *'a projection* → *'a* → *'a*
cast ≡ Λ*D*. *Rep-projection D*

in-projection :: *'a* ⇒ *'a projection* ⇒ *bool* (**infixl** ::: *50*)
x ::: *D* ≡ *cast*·*D*·*x* = *x*

lemma *cast-in-projection*: *cast*·*D*·*x* ::: *D*
lemma *subprojectionD*: ⟦*D* ⊑ *E*; *x* ::: *D*⟧ ⟹ *x* ::: *E*

3.2 Representable Types

Using the domain package of Isabelle/HOLCF, we can define a universal domain U that is isomorphic to the lifted naturals plus its own continuous function space:

domain *U* = *UNat* (*fromUNat* :: *nat lift*) | *UFun* (*fromUFun* :: *U* → *U*)

The domain package defines continuous constructor and accessor functions for the type U, and proves a collection of theorems about them. Then we can easily show that *UNat* and *fromUNat* form an ep-pair from *nat lift* to U; similarly, *UFun* and *fromUFun* form an ep-pair from $U \to U$ to U. Using these ep-pairs as a starting point, we can then construct ep-pairs to U for several other types: *unit lift*, *bool lift*, $U \times U$, $U \otimes U$, $U \oplus U$, and $U_\bot$.

We say that a type $'a$ is *representable* if we can define an ep-pair between $'a$ and the universal domain U. We declare overloaded constants *emb* and *proj* to convert values to and from the universal domain, and encode the notion of representability with a new axiomatic type class *rep*:

consts
emb :: $'a \to U$
proj :: $U \to 'a$

axclass *rep* $\subseteq$ *pcpo*
ep-emb-proj: *is-ep-pair emb proj*

Given two representable types $'a$ and $'b$, we can lift the ep-pairs over the cartesian product to construct a new ep-pair from $'a \times 'b$ to $U \times U$. Then we can compose this with the standard ep-pair from $U \times U$ to U, and get an ep-pair from $'a \times 'b$ to U. Thus $'a \times 'b$ is representable if both $'a$ and $'b$ are, and we say that $\times$ is a *representable type constructor*. Similarly, we can show that $\to$, $\otimes$, $\oplus$, and $(-)_\perp$ are all representable type constructors as well. Our proofs of representability are analogous to those given by Gunter and Scott [7, §7.1, §7.3]. Note that U is also trivially representable, since we can embed any type into itself.

Mapping from Types to Values. We encode *values* of representable types as *values* of type U, but we encode *types* themselves as *projections* over U. We can construct a projection for any representable type by simply composing *emb* and *proj*.

constdefs
rep-of :: $('a$::*rep*$)$ *itself* $\Rightarrow$ *U projection*
rep-of $(t$::$'a$ *itself*$) \equiv$ *Abs-projection* $((emb$::$'a \to U) \; oo \; (proj$::$U \to 'a))$

The special type $'a$ *itself* has only one value, which is written with the special syntax $TYPE('a)$. As the argument type of *rep-of*, it allows us to effectively take types as arguments. We could have used simply $'a$ as the argument type, but this does not accurately reflect what the function does—the result value does not depend on any actual values of type $'a$, it only depends on the type itself. This Isabelle-specific feature is not actually necessary: We could also have defined a type constructor that produces singleton types, using the datatype package; instead of $TYPE('a)$ we would just write *Myself*::$'a$ *itself*.

datatype $'a$ *itself* = *Myself* — *sample singleton type constructor*

The type *U projection* is large enough to encode all the programming language datatypes that we are interested in. We have already shown that the unit type, sums, products, and continuous function spaces are representable, and since *U projection* is a pcpo, it contains least fixed-points for all general recursive datatypes as well.

From now on, all free type variables are assumed to be in class *rep*, unless specified otherwise. We will not concern ourselves with non-representable types.

3.3 Representing Type Constructors

One way to think of a type constructor is as a function from types to types. Just as we can represent a type with a projection, we can represent a type constructor using a projection constructor, i.e. a continuous function of type *U projection* → *U projection*.

We declare an Isabelle type class *tycon* for type constructors. It is a syntactic class which has no axioms; the class only serves to restrict the possible argument types for overloaded functions. While instances of class *tycon* are actually types, we will never use them as such, or construct any values having those types. We only use them to define an overloaded constant *tc*, which returns a projection constructor. As before, we use *'f itself* as the argument type because the result should only depend on the type of the argument, and not its value.

axclass *tycon* ⊆ *type*
consts *tc* :: (*'f*::*tycon*) *itself* ⇒ *U projection* → *U projection*

Now we can define an Isabelle type constructor to model explicit type application. *App* takes two type arguments: *'a* of class *rep*, and *'f* of class *tycon*. By applying the projection constructor of *'f* to the representation of *'a*, we get a new projection. We define the resulting type to be isomorphic to the subset of *U* that corresponds to this projection.[1]

typedef (**open**) (*'a*,*'f*) *App* (**infixl** ⊙ *65*)
= {*x*. *x* ::: *tc TYPE*(*'f*::*tycon*)·(*rep-of TYPE*(*'a*::*rep*))}

defs
emb :: *'a*⊙*'f* → *U* ≡ Λ *x*. *Rep-App x*
proj :: *U* → *'a*⊙*'f* ≡ Λ *x*. *Abs-App* (*cast*·(*tc TYPE*(*'f*)·(*rep-of TYPE*(*'a*)))·*x*)

lemma *rep-of-App*:
rep-of TYPE(*'a*⊙*'f*) = *tc TYPE*(*'f*)·(*rep-of TYPE*(*'a*))

Since λ*x*. *x* ::: *D* is an admissible predicate, and holds for ⊥, it is easy to show that the resulting type is a pcpo. Also notice the infix syntax for the *App* type constructor: We use postfix application order to be consistent with Isabelle's type syntax.

4 Axiomatic Constructor Classes

4.1 Coercion

We define a function *coerce* to convert values between any two representable types. The *coerce* function will be useful in the next section for defining polymorphic constants in axiomatic type classes.

[1] For convenience, Isabelle's typedef package would normally try to define a constant *App* to be equal to the set specified in the type definition. That would cause an error in this case, because while the value of the set depends on the type variables *'a* and *'f*, those type variables do not appear in the type of the set itself. The keyword **open** overrides this behavior, so that the typedef package does not define such a constant.

constdefs
coerce :: ('*a*::*rep*) → ('*b*::*rep*)
coerce ≡ *proj oo emb*

Now we establish some properties of *coerce*. Coercing from a smaller type to a larger type is always invertible, while coercing from a larger type to a smaller type is only invertible under some conditions. Applying a coerced function is equivalent to coercing before and after applying the original function. Finally, the *coerce* function may degenerate into *emb*, *proj*, or *ID*, depending on the type at which it is applied.

lemma *coerce-inv1*: *rep-of TYPE*('*a*) ⊑ *rep-of TYPE*('*b*)
⟹ *coerce*·(*coerce*·*x* :: '*b*) = (*x*::'*a*)
lemma *coerce-inv2*: *emb*·*x* ::: *rep-of TYPE*('*b*) ⟹ *coerce*·(*coerce*·*x* :: '*b*) = (*x*::'*a*)
lemma *coerce-cfun*: *coerce*·*f* = *coerce oo f oo coerce*
lemma *coerce-ID*: *coerce*·(*ID*::'*a* → '*a*) = *cast*·*rep-of TYPE*('*a*)

4.2 Functor Class

The first axiomatic type class that we define is the *functor* class. Instances of this class should have a polymorphic function *fmap* (similar to *map* for lists) that preserves the identity and function composition. Here are the type signature and theorems that we would like to have for *fmap*:

consts *fmap* :: ('*a* → '*b*) → '*a*⊙'*f* → '*b*⊙('*f*::*functor*)
theorem *fmap-ID*: *fmap*·*ID* = *ID*
theorem *fmap-comp*: *fmap*·(*f oo g*) = *fmap*·*f oo fmap*·*g*

The above theorems are not suitable for use as class axioms, because they each have multiple free type variables: *fmap-ID* has two and *fmap-comp* has four, counting '*f*. We could try to emulate a 4-parameter type class using predicates, but when reasoning about functors in general the number of extra assumptions would quickly get out of hand. We really need a set of class axioms with only *one* free type variable, the type constructor '*f*. Furthermore, the class axioms should ensure that the above laws hold at all instances of the other type variables. If Isabelle supported nested universal quantification of type variables [13] this would be simple to express, but we must find another way.

Our solution is to express the functor laws in an untyped setting, by replacing universally-quantified type variables with *U*, the universal domain type. In this setting we can model type quantification using quantification over the *U projection* type. Along these lines, we declare an "untyped" version of *fmap* called *rep-fmap*—the polymorphic *fmap* is defined by coercing it to more specific types.

consts *rep-fmap* :: (*U* → *U*) → *U*⊙'*f* → *U*⊙'*f*
defs *fmap* ≡ *coerce*·*rep-fmap*

The *functor* class axioms are all in terms of *rep-fmap*. We need three altogether: one for each of the two functor laws, and one to assert that *rep-fmap* has

an appropriate polymorphic type. We just need to decide what forms the laws should take. First consider the composition law *fmap-comp*: If we convert it to the untyped setting and make all quantification explicit, we get something like the following (abusing notation slightly):

theorem *rep-fmap-comp*:
$\forall\, a\ b\ c{::}U\ \mathit{projection}.$
$\forall f{:::}b{\to}c.\ \forall g{:::}a{\to}b.\ \mathit{rep\text{-}fmap}{\cdot}(f\ \mathit{oo}\ g) = \mathit{rep\text{-}fmap}{\cdot}f\ \mathit{oo}\ \mathit{rep\text{-}fmap}{\cdot}g$

Because the *U projection* variables are only used to restrict the quantification of f and g, we can simplify this rule by removing the *U projection* quantifications, and allowing f and g to range over all functions of type $U \to U$. We end up with something that looks exactly like the original *fmap-comp* law, but at a more specific type.

The identity law *fmap-ID* works out differently, because it mentions a specific function value *ID* instead of using universal quantification. When we convert this rule to the untyped setting, we obtain terms of the form *coerce*·*ID*, which simplify to applications of *cast*.

axclass *functor* $\subseteq$ *tycon*
rep-fmap-type:
$[\![\bigwedge x.\ x :::\ D \Longrightarrow f{\cdot}x :::\ E;\ \mathit{emb}{\cdot}\mathit{xs} :::\ \mathit{tc}\ \mathit{TYPE}('f){\cdot}D]\!]$
$\Longrightarrow \mathit{emb}{\cdot}(\mathit{rep\text{-}fmap}{\cdot}f{\cdot}\mathit{xs}) :::\ \mathit{tc}\ \mathit{TYPE}('f){\cdot}E$

rep-fmap-ID:
$\mathit{rep\text{-}fmap}{\cdot}(\mathit{cast}{\cdot}D) = \mathit{proj}\ \mathit{oo}\ \mathit{cast}{\cdot}(\mathit{tc}\ \mathit{TYPE}('f){\cdot}D)\ \mathit{oo}\ \mathit{emb}$

rep-fmap-comp:
$\mathit{rep\text{-}fmap}{\cdot}(f\ \mathit{oo}\ g) = \mathit{rep\text{-}fmap}{\cdot}f\ \mathit{oo}\ \mathit{rep\text{-}fmap}{\cdot}g$

Deriving the polymorphic versions of the functor laws is quite straightforward. It basically consists of unfolding the definition of *fmap*, and then using the *coerce* lemmas together with the *functor* class axioms to finish the proofs.

4.3 Functor Class Instances

In this section we describe the recommended method for establishing instances of the functor class. In typical usage, we expect the user to have defined an ordinary Isabelle type constructor—such as *llist*, for lazy lists—which is representable and has a function *map* that satisfies the functor laws. Our remaining task is to define a *tycon* instance that models this type constructor, and prove that it is an instance of the functor class.

To obtain the projection constructor that models *llist*, we need to derive a formula for *rep-of* $\mathit{TYPE}('a\ \mathit{llist})$ in terms of *rep-of* $\mathit{TYPE}('a)$. This is straightforward as long as *emb* and *proj* are defined in a reasonable way for *llist*: Embedding a value of type $'a\ \mathit{llist}$ should be the same as first mapping *emb* over the list, and then embedding the resulting *U llist*. Similarly, projecting a value of type $'a\ \mathit{llist}$ should be the same as projecting from U to *U llist*, and then applying *map*·*proj*.

rep-of TYPE('a llist)
  = *Abs-projection* (*emb oo proj*)
  = *Abs-projection* (*emb oo map·emb oo map·proj oo proj*)
  = *Abs-projection* (*emb oo map·*(*emb oo proj*) *oo proj*)
  = *Abs-projection* (*emb oo map·*(*cast·*(*rep-of TYPE*('*a*))) *oo proj*)

Accordingly, we define a function *functor-tc* that produces a projection constructor from any map function. We will typically instantiate the type variable $'l$ to some type constructor applied to U, for example *U llist*.

constdefs
  functor-tc :: ((*U* → *U*) → '*l* → '*l*) ⇒ *U projection* → *U projection*
  functor-tc map ≡ Λ*D*. *Abs-projection* (*emb oo map·*(*cast·D*) *oo proj*)

Next we declare a type which will be an instance of class *tycon*. Since values of types in class *tycon* are never used for anything, it does not matter how we actually define the type—a singleton type works fine. By our convention, we use a capitalized version of the name of the original type constructor.

datatype *LList* = *DummyLList* — *exact definition does not matter*
instance *LList* :: *tycon* ..
defs
  tc (*t*::*LList itself*) ≡ *functor-tc* (*map*::(*U* → *U*) → *U llist* → *U llist*)
  rep-fmap::(*U* → *U*) → *U*⊙*LList* → *U*⊙*LList* ≡ *coerce·map*

The above two definitions, together with proofs of the functor laws for *map* at type (*U* → *U*) → *U llist* → *U llist*, are sufficient to prove the functor class axioms for type *LList*. To facilitate proof reuse, we define a predicate to encode this set of assumptions.

constdefs
  functor-locale :: ('*f*::*tycon*) *itself* ⇒ ((*U* → *U*) → '*l* → '*l*) ⇒ *bool*
  functor-locale (*t*::'*f itself*) *map* ≡
    (*tc TYPE*('*f*) = *functor-tc map*) ∧
    ((*rep-fmap* :: (*U* → *U*) → *U*⊙'*f* → *U*⊙'*f*) = *coerce·map*) ∧
    (*map·ID* = *ID*) ∧
    (∀ *f g*. *map·*(*f oo g*) = *map·f oo map·g*)

Using only this predicate as an assumption, we can prove each of the functor class axioms. There are a couple of important intermediate theorems: First, that the argument to *Abs-projection* in the definition of *functor-tc* is actually a projection. Second, that the types *U*⊙'*f* and '*l* are represented by the same projection—this means *coerce* is an isomorphism between the two. Using these lemmas together with the properties of *coerce*, it is then relatively straightforward to prove the three functor class axioms for type '*f*. This means that to establish a functor class instance, the user only needs to define the *tycon* and *rep-fmap* in the standard way, and then prove that the functor laws hold at a single type instance.

To show that *LList* is an instance of class *functor*, our only proof obligation is to show *functor-locale TYPE*(*LList*) *map*. Once we have shown that *LList* is in class functor, we can now use *fmap* at type $('a \to 'b) \to 'a\odot LList \to 'b\odot LList$. We can also instantiate the functor laws *fmap-ID* and *fmap-comp* (or any other theorem proved about functors in general) to the *LList* type. Besides *fmap*, there are no constants or operations defined on the new *LList* type; however, we can always create values or operations on the type $'a\odot LList$ by coercion from the type *'a llist*, since the two types are isomorphic.

4.4 Monad Class

The *monad* class specifies two polymorphic constants, *return* and *bind*, with a set of three monad laws that they should satisfy. In addition, a fourth law should be satisfied by monads that are also instances of the *functor* class. Here are the type signature and theorems that we would like to have for class *monad*.

consts *return* :: $'a \to 'a\odot('m::monad)$
consts *bind* :: $'a\odot('m::monad) \to ('a \to 'b\odot'm) \to 'b\odot'm$ (**infixl** ▷ *55*)
theorem *monad-left-unit*: $(return{\cdot}x \triangleright f) = (f{\cdot}x)$
theorem *monad-right-unit*: $(m \triangleright return) = m$
theorem *monad-bind-assoc*: $((m \triangleright f) \triangleright g) = (m \triangleright (\Lambda\ x.\ f{\cdot}x \triangleright g))$
theorem *monad-fmap*: $fmap{\cdot}f{\cdot}xs = xs \triangleright (\Lambda\ x.\ return{\cdot}(f{\cdot}x))$

The functor and monad laws are closely connected. If we use the *monad-fmap* law as a definition for *fmap*, then the functor laws can be proved from the monad laws. Alternatively, we can define *monad* as a subclass of *functor*; in this case, the right unit law is redundant. We have chosen the subclass method, because it allows us to reuse some proofs from the *functor* class.

The definition of the *monad* class follows the same basic pattern as the *functor* class. We start by declaring overloaded representative versions of *return* and *bind*, where all polymorphic type variables are replaced with *U*. As with the functor class, the real *return* and *bind* are defined by coercion from these.

The *monad* class has five axioms in total: one each for the types of return and bind, and three more for the *monad-left-unit*, *monad-bind-assoc*, and *monad-fmap* rules. All three of these class axioms look exactly like the original rules, because (as with *fmap-comp*) all free type variables are attached to universally quantified values.

Our standard method for establishing instances of class *monad* is similar to the method for class *functor*. We define a predicate *monad-locale* that encodes all of the assumptions necessary to prove the *monad* class axioms. This predicate includes all the assumptions from *functor-locale*, plus additional assumptions stating that *rep-bind* and *rep-return* are defined by coercion, and that the monad laws hold for the underlying type.

4.5 Monad Transformers

In addition to simple type constructors like *LList*, our framework can also be used to define type constructors that take additional type arguments, some of

which may be type constructors themselves. The trick is to use a type constructor with one or more type arguments as an instance of class *tycon*.

A good example is the state monad transformer, which has a total of three type parameters: The state type $'s$, the inner monad $'m$, and the result type $'a$. We declare $('a,'m,'s)$ *stateT* as a type abbreviation for $'s \to ('a \times 's)\odot'm$, and define operations *map-stateT*, *return-stateT* and *bind-stateT* on this type in terms of the monad operations of $'m$. We can then use the monad laws for $'m$ to prove that these new operations satisfy the monad laws.

datatype $('m,'s)$ *StateT* = *DummyStateT* — *exact definition does not matter*
instance *StateT* :: (*monad*, *rep*) *tycon* ..
defs *tc* (*t*::$('m,'s)$ *StateT itself*)
≡ *functor-tc* (*map-stateT*::$(U \to U) \to (U,'m,'s)$ *stateT* $\to (U,'m,'s)$ *stateT*)

The above instance declaration says that $('m,'s)$ *StateT* is in class *tycon* if $'m$ is in class *monad* and $'s$ is in class *rep*. After defining *tc*, *rep-fmap*, *rep-return*, and *rep-bind* in the standard way, we can then use the *monad-locale* theorems to establish that $('m,'s)$ *StateT* is also in class *monad*, for any monad $'m$.

instance *StateT* :: (*monad*, *rep*) *monad*

A more complex example is the resumption monad transformer: It is particularly interesting because the datatype is defined with indirect recursion through a monad parameter. The domain package of HOLCF can not define such datatypes; we defined it manually by taking the least fixed-point of a projection constructor. In the definition of *rep-resT*, $(-)_\perp$ and $\oplus$ refer to the projection constructors associated with their respective type constructors.

constdefs
rep-resT :: (*U projection* $\to$ *U projection*) $\to$ *U projection* $\to$ *U projection*
rep-resT ≡ $\Lambda M\ A.$ *fix*·$(\Lambda R.\ A_\perp \oplus (M{\cdot}R)_\perp)$

typedef $('a,'m)$ *resT* = $\{x.\ x :::$ *rep-resT*·(*tc* *TYPE*$('m)$)·(*rep-of* *TYPE*$('a)$)$\}$

The resulting type satisfies the isomorphism $('a,'m)$ *resT* $\cong {'a}_\perp \oplus (('a,'m)$ *resT*$\odot'm)_\perp$. Similarly to the previous example, we define a type $'m$ *ResT* as a member of class *tycon*, and we prove that if $'m$ is in class *monad*, then so is $'m$ *ResT*.

5 Axiomatic Constructor Classes in HOL

All of the framework described so far has been implemented in Isabelle/HOLCF, where every representable type is a pcpo. However, we have also explored the possibility of porting the theories to Isabelle/HOL. Most of it appears to work, albeit with some important restrictions. The major differences between the two versions are summarized in Table 1.

The essential characteristics of the class of representable types are that: (1) The class is closed with respect to several type constructors, and (2) the class

Table 1. Translation between HOLCF and HOL versions of axiomatic constructor classes

Concept	HOLCF	HOL
Representable type constructors	$\rightarrow, \times, \otimes, \oplus, \cdot_\perp$	$\times, +$, restricted $\Rightarrow$
Embedding between types	ep-pair	function with inverse
Encoding of representable type	projection over U	idempotent function
Ordering of type encodings	$\sqsubseteq$ on projections	$\subseteq$ on range sets
Encoding of type constructor	continuous function	monotone function

has an ordering, with a maximal representable type U. Having a maximal representable type is necessary for our method of reasoning about polymorphic constants in constructor classes.

Remember that in HOLCF we can define a universal domain U into which we can embed its continuous function space $U \rightarrow U$. However, this is not possible in HOL, since for any non-trivial type U the full function space $U \Rightarrow U$ has a strictly larger cardinality than U itself. Essentially this means that the full function space type constructor $\Rightarrow$ can not be representable.

It is possible to make $\Rightarrow$ representable in a limited way, by placing extra restrictions on the left type argument. For example, $'a \Rightarrow 'b$ could be representable for all representable types $'b$ and *countable* types $'a$. Isabelle's datatype package can define infinitely-branching trees, which would be a good candidate for a universal type that could represent these function spaces.

6 Related Work

The authors are members of the Programatica project [10], which is building a high assurance software development environment for Haskell98 [11]. Programatica allows users to embed desired correctness assertions and environmental assumptions in Haskell program elements. Assertions can also be annotated with *certificates* that provide evidence of validity, at differing levels of assurance. Example certificates can range from code inspection sign-offs, manually or randomly generated test cases, all the way up to theorem proving invocations. The Programatica environment tracks assertions and the Haskell definitions they depend on, and can re-invoke certificate servers automatically as needed. We are using Isabelle/HOLCF and axiomatic constructor classes to build a certificate server that provides a high level of assurance for validating Programatica assertions, even in the presence of Haskell functions that terminate on only a subset of their inputs.

Papaspyrou and Macos [19] illustrate how monads and monad transformers can provide a modular denotational semantics for a range of programming language features. They define a simple eager language of expressions with side effects (ELSE), and gradually extend the semantics to include side effects, nondeterministic evaluation of side-effects, concurrent execution of side-effects, and culminates in ANSI C style sequence points. The semantics of the language is

parameterized on an underlying monad; this allows the desired computational effects to be reconfigured without globally rewriting the language semantics. The desired monad is also constructed modularly, by applying a sequence of monad transformers. The same framework can also be adapted to model structural language features such as procedures, non-local control-flow, and reference values. Papaspyrou has given a denotational semantics for a significant subset of ANSI C using the same methods [18].

Proof assistants such as Coq [4] and MetaPRL [8] whose type systems are based on dependent type theory can encode instances of axiomatic type and constructor classes as records whose fields contain implementations of the class methods, as well as proofs that the methods satisfy the class axioms. Overloaded functions can then be defined as functions that take these records as parameters. This is often called the *dictionary-passing* approach to implementing type classes. It is unclear to us whether the function parameter hiding and type inference heuristics of these proof assistants are sufficient to hide such dictionary passing from the user, as is the case with our implementation of constructor classes in Isabelle/HOLCF.

Theory morphisms are an alternative to axiomatic type classes for allowing theorems to be reused across families of types, and have been implemented in theorem provers such as IMPS [6,5] and PVS [17]. A key advantage of theory morphisms is that multiple morphisms can be defined that target the same type.

Axiomatic constructor classes can be simulated by theory morphisms, provided that the morphism is allowed to instantiate type constructors of arity greater than zero. However, instantiation of morphisms is an operation on theories, rather than terms, and therefore cannot be applied anonymously to subterms. Also, most-general class instantiations for a well-typed term can always be inferred by Isabelle's order-sorted type unification algorithm [15]. Larger Haskell programs rely on this heavily, and it prevents type annotations from swamping the actual code. To our knowledge, no similar capability is available for current theory morphism implementations. However, they could be implemented in principle, if one were willing to specify a "default" morphism for any given type scheme in the same way that class instances are defined.

Isabelle has a lightweight implementation of theory morphisms, called *locales* [3,12]. However, locales can not not instantiate type constructors, so they are unsuitable for modeling constructor classes. A more general theory morphism mechanism has recently been implemented for Isabelle by Johnsen and Lüth [9], that relies on the theorem prover's ability to attach proof objects to theorems. This allows theorems to be safely instantiated, without needing to modify Isabelle's kernel.

7 Conclusion

Using purely definitional means, we have developed a framework within Isabelle/HOLCF that permits abstract reasoning about type constructors. We have formalized the functor and monad type classes and proved several monad

instances, including the maybe monad, lazy lists, the error monad, and the state monad. We have also formalized monad transformers for error handling, persistent state, and resumptions.

We have found that our framework works quite well for abstract reasoning about functors and monads in general. Isabelle's type class system neatly encapsulates all the assumptions related to the functor and monad laws.

Our framework still has much room for improvement, though. Even with a library of combinators available, it turns out that constructing *emb* and *proj* functions takes a bit of work for recursive types; this is something that would benefit from automation. It is also unfortunate that in our framework, we end up with two versions of each type constructor, for example *llist* and *LList*. This means that constants and theorems about *llist* must all be transferred over to *LList* one by one. This would benefit from automation as well. Alternatively, it would be nice to have a datatype package that generates *tycon* instances in the first place.

Other directions for future work aim to automate the translation from Haskell-style code into Isabelle definitions. Mechanizing the process of producing Isabelle code for new type classes is one possibility. With a more sophisticated universal domain, it may also be possible to model datatypes that use features like higher-rank polymorphism, which would be valuable for deep embeddings of Haskell semantics.

References

1. S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.
2. Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda Calculi*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
3. Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In *Types for Proof and Programs, Intl. Workshop TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.
4. Yves Bertot and Pierre Castéran. Interactive theorem proving and program development. Coq'Art : The calculus of inductive constructions. Texts in Theoretical Computer Science, page 492, 2004.
5. W. M. Farmer. An infrastructure for intertheory reasoning. In D. McAllester, editor, *Automated Deduction—CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 115–131. Springer-Verlag, 2000.
6. W. M. Farmer, J. D. Guttman, and F. J. Thayer Fábrega. IMPS: An updated system description. In M. McRobbie and J. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 1996.
7. C. A. Gunter and D. S. Scott. Semantic domains. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 633–674. MIT Press, 1990.

8. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. pages 287–303.
9. Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 152–167. Springer, September 2004.
10. Mark P. Jones. Programatica web page, 2005. `http://www.cse.ogi.edu/PacSoft/projects/programatica`.
11. Simon Peyton Jones. *Haskell98 Language and Libraries, Revised Report*, December 2002.
12. Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *12th International Conference on Theorem Proving in Higher-Order Logics TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*. Springer, September 1999.
13. Thomas F. Melham. The HOL logic extended with quantification over type variables. In *HOL'92: Proceedings of the IFIP TC10/WG10.2 Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 3–17. North-Holland/Elsevier, 1993.
14. Olaf Müller, Tobias Nipkow, David Von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.
15. Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.
16. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle / HOL, A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
17. S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.
18. N.S. Papaspyrou. *A formal semantics for the C programming language*. PhD thesis, National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, February 1998.
19. N.S. Papaspyrou and D. Macos. A study of evaluation order semantics in expressions with side effects. *Journal of Functional Programming*, 10(3):227–244, May 2000.
20. Franz Regensburger. HOLCF: Higher order logic of computable functions. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 293–307. Springer-Verlag, 1995.
21. Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 307–322. Springer-Verlag, 1997.

Meta Reasoning in ACL2

Warren A. Hunt Jr., Matt Kaufmann, Robert Bellarmine Krug,
J Strother Moore, and Eric Whitman Smith

Department of Computer Sciences, University of Texas at Austin,
Austin, TX 78712-1188, USA
{hunt, kaufmann, rkrug, moore}@cs.utexas.edu
ewsmith@stanford.edu

Abstract. The ACL2 system is based upon a first-order logic and implements traditional first-order reasoning techniques, notably (conditional) rewriting, as well as extensions including mathematical induction and a "functional instantiation" capability for mimicking second-order reasoning. Additionally, one can engage in meta-reasoning — using ACL2 to reason, and prove theorems, about ACL2's logic from within ACL2. One can then use these theorems to augment ACL2's proof engine with custom extensions. ACL2 also supports forms of meta-level control of its rewriter. Relatively recent additions of these forms of control, as well as extensions to ACL2's long-standing meta-reasoning capability, allow a greater range of rules to be written than was possible before, allowing one to specify more comprehensive proof strategies.

1 Introduction

ACL2 is a logic, a programming language, and a semi-automatic theorem prover [6,4,5]. This paper is about the meta reasoning facilities of ACL2, the theorem prover. We give a brief overview of ACL2's operations, paying particular attention to ACL2's rule-based rewriter, which is generally considered to be its main proof procedure. We then present a sequence of increasingly complex problems that cannot be solved with normal rewrite rules and show how they can be solved using ACL2's meta-reasoning facilities. Although occasionally simplified, all but one of these examples are based upon items from actual proof efforts. One can find these as well as many additional examples in the proof scripts distributed with ACL2 by searching for the keywords syntaxp, meta, and bind-free.

The facilities presented in this paper correspond to "computational reflection" as described by Harrison [3]. We do not, however, wish to argue with his thesis that there is often a "too-easy acceptance of reflection principles as a practical necessity." Rather, we argue that these facilities, when carefully integrated into a system such as ACL2, can greatly enhance the user experience. Although ACL2 does have a tactic programming language — for its interactive utility (the so-called *proof-checker*) — experience has shown that the techniques described here are often simpler and more profitable to use.

The facilities described in this paper fall into three categories: meta functions (dating back to [1]), syntaxp directives, and bind-free directives. In addition, each

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 163–178, 2005.

of these three facilities can be divided into a "plain" and an "extended" version. The plain version of meta functions has been present in ACL2 from its inception, and syntaxp was added not long thereafter (in the early 1990s). Extended meta functions were added in Version 2.6 (2001). The bind-free directive was added to Version 2.7 (2002), in both its plain and extended versions. The extended syntaxp directive was added at the same time.

Syntaxp and bind-free might be appropriate within an LCF-style system such as HOL [2]. We do not intend to argue that point vigorously, although it seems that they could serve to free users from the need to work in the tactic meta-language.[1] We do believe that these facilities bring many of the benefits of HOL's programmability to ACL2. Additionally, even experienced ACL2 users may find it simpler to add a syntaxp or bind-free directive than to prove a *meta rule* that installs a new simplification procedure.

We begin with a few words about the ACL2 language. The ACL2 language is based upon a subset of Common Lisp. As such, it uses a prefix notation. For example, one might write an expression `3*f(x,y+3)` in a traditional notation, or in the C programming language, which would be written in Lisp notation as `(* 3 (f x (+ y 3)))`. We should also mention that ACL2 terms are themselves objects, which therefore can be constructed and analyzed by ACL2. Without this ability, the features we describe in this paper would not have been possible. In this paper, however, we stick with a traditional (or C language) notation for pedagogical purposes.

We will also use function names that are self-explanatory. For example, the ACL2 term `(quotep x)`, which we could write as `quotep(x)`, would be written in this paper as `constant(x)` (so that we need not explain that `quotep` is the recognizer for constant terms). We will also avoid Lisp's quote notation by writing, for example, `fn-symb(x) == +` to indicate that the top function symbol of the term `x` is the symbol `+`, in place of the ACL2 notation `(equal (fn-symb x) '+)`. We therefore assume no familiarity with Lisp on the part of the reader, yet with the comments above we also expect it to be readable by those familiar with Lisp or ACL2.

We now give our brief overview of ACL2. The user submits a purported theorem to ACL2, which applies a series of procedures in an attempt to prove the theorem. These procedures include, among others, the following: simplification that includes rewriting and linear arithmetic; generalization; and induction. ACL2 is fully automatic in the sense that this process, once started, cannot be further guided.

That said, however, ACL2 will rarely succeed at proving any but the simplest of theorems on its own. The user usually must assist ACL2 in finding a proof, generally by attaching hints to a theorem or by proving additional rules. New rules will be added to ACL2's database and used in subsequent proof attempts.

[1] We thank a referee for pointing out that Isabelle's Isar user interface is an example of a "recent trend in some higher order logic theorem provers to shield users from having to learn the tactic meta-language."

The judicious development of a library of rules can make ACL2 not only powerful but — in the words of a game review — strangely glee to play with.

ACL2's rewriter uses conditional rewrite rules and proceeds in a left-to-right, inside-out manner, maintaining a context as it goes. This will be clearer after an example. When ACL2 is rewriting a goal of the form:

```
    <hyp 1> &&
    <hyp 2>
==> <from> == <to>
```

it acts as if it were rewriting the equivalent clause (i.e., disjunction, represented here using `||`)

```
not(<hyp 1>) || not(<hyp 2>) || (<from> == <to>)
```

and attempts to rewrite, in (left-to-right) order, each of the following to true: `not(<hyp 1>)`, `not(<hyp 2>)`, and `(<from> == <to>)`. As ACL2 rewrites each disjunct above, it does so in a context in which it assumes the falsity of the others. Thus, when ACL2 rewrites `not(<hyp 1>)`, it assumes both `<hyp 2>` and `(<from> != <to>)`. Similarly, when it rewrites `(<from> == <to>)`, it assumes rewritten forms of `<hyp 1>` and `<hyp 2>`. These assumptions are the context.

Suppose ACL2 is rewriting a function application, say `foo(<arg1>, <arg2>)`. In this case, ACL2 will recursively rewrite (proceeding left to right) each of `<arg1>` and `<arg2>`, yielding `<arg1'>` and `<arg2'>` and then rewrite the expression `foo(<arg1'>, <arg2'>)`. Note that this inside-out rewriting order mimics that for evaluation — a function's arguments are evaluated before the function is applied to them[2].

Let us now examine the rewriter in more detail, and see how this last expression, `foo(<arg1'>, <arg2'>)`, may be rewritten. Assume that the conditional rewrite rule

```
GIVEN p(x)
      q(y)
REWRITE foo(x, y) TO bar(y, x)
```

had been previously proved. The left-hand side of the above conclusion — `foo(x, y)` — can be matched with our target term — `foo(<arg1'>, <arg2'>)` — by replacing the variables `x` with `<arg1'>` and `y` with `<arg2'>`. Instantiating the above theorem thus yields:

```
GIVEN p(<arg1'>)
```

[2] And as for evaluation, the rewriter handles if-then-else terms in a "lazy" manner: in order to rewrite the term

```
if <test> then <true-branch> else <false-branch>
```

ACL2 first rewrites the test, and if the result is true or false then ACL2 rewrites only the true branch or the false branch, respectively. Otherwise the resulting `if` term will generally lead, ultimately, to a case split.

```
      q(<arg2'>)
REWRITE foo(<arg1'>, <arg2'>) TO bar(<arg2'>, <arg1'>).
```

If ACL2 can relieve the hypotheses — recursively rewrite them to true — it will replace the expression `foo(<arg1'>, <arg2'>)` with `bar(<arg2'>, <arg1'>)`.

Conditional rewriting, as illustrated above, is quite restrictive. This paper presents techniques that allow a much greater range of replacements to be specified — they allow one to specify solutions to classes of problems and to experiment speculatively with several rewriting strategies, selecting among these based upon the predetermined outcome of these strategies. Note, in particular, that we are not claiming that these facilities allow us to prove things we could not, in principle, before. Rather, we developed these facilities to help the user to prove theorems more easily and naturally, by removing much of the tedium of repeatedly carrying out simple and "obvious" steps.

2 Syntaxp

When reasoning about arithmetic expressions, it is usual to have some rules like the following to assist with normalizing sums:

```
RULE: associativity-of-+
REWRITE (x + y) + z TO x + (y + z)

RULE: commutativity-of-+
REWRITE y + x TO x + y

RULE: commutativity-2-of-+
REWRITE y + ( x + z) TO x + (y + z)
```

Although it may appear that the second and third rules could each loop or be applied repeatedly, they permute the individual summands into a pre-defined term-order.[3] Rules that merely permute their elements without introducing any new function symbols, such as the aforementioned two rules, are recognized by ACL2 as potentially looping. It will apply such rules only when doing so will move a smaller term to the left of a larger one. Thus, for instance, ACL2 will use `commutativity-of-+` to rewrite `x + 3`, `y + x`, and `(y + z) + x` to `3 + x`, `x + y`, and `x + (y + z)` respectively, but will not apply it to any of these latter expressions.

Note that although there was no meta-level reasoning used to justify these rules, and although there were no meta-level heuristics explicitly given by the user, the behavior of `commutativity-of-+` and `commutativity-2-of-+` are restricted based upon the syntactic form of the instantiations of the variables `x`, `y`, and `z`.

[3] The details of this term-order are irrelevant to the present paper; but, crudely, it is a lexicographic order based upon the number of variables, the number of function symbols, and an alphabetic order.

Let us now consider the term `x + (3 + 4)`. Recall that ACL2 rewrites inside-out. Thus, ACL2 will first rewrite the two arguments, `x` and `(3 + 4)`. The first of these is a variable, and so rewrites to itself. The second of these is a ground term and, since ACL2 implements an executable logic, this term will get evaluated to produce `7`. Finally, ACL2 will use `commutativity-of-+` to rewrite `x + 7` to `7 + x`.

But what about `3 + (4 + x)`? Ideally, this would rewrite to the same thing, but there is nothing the above rules can do with this. If we could only get the `3` and `4` together, things would proceed as for `x + (3 + 4)`. The following rule will do this for us:

```
RULE: fold-consts-in-+
GIVEN syntaxp(constant(c))
      syntaxp(constant(d))
REWRITE c + (d + x) TO (c + d) + x.
```

This rule is just the reverse of `associativity-of-+`, with the addition of two syntaxp hypotheses. Without these extra hypotheses, this rule would loop with `associativity-of-+`.

How do these syntaxp hypotheses work? Logically, a syntaxp expression evaluates to true. The above rule is, therefore, logically equivalent to

```
GIVEN t
      t
REWRITE c + (d + x) TO (c + d) + x
```

or

```
REWRITE c + (d + x) TO (c + d) + x
```

and this is the meaning of `syntaxp` when one is proving the correctness of a rule. (Note that `t` denotes true.)

However, when attempting to apply such a rule, the test inside the syntaxp expression is treated as a meta-level statement about the proposed instantiation of the rule's variables, and that instantiated statement must evaluate to true to establish the syntaxp hypothesis. Note, in particular, that the statement must *evaluate* to true, rather than be proved true as for a regular hypothesis. Thus, just as term-order is automatically used as a syntactic restriction on the operation of `commutativity-of-+` and `commutativity-2-of-+`, so we have placed a syntactic restriction on the behavior of `fold-consts-in-+` — the variables `c` and `d` must be matched with constants.

Here, we are considering the application of `fold-consts-in-+` to the term `3 + (4 + x)`. The variable `c` of the rule is matched with `3`, `d` with `4`, and `x` with `x`. Since `3` and `4` are, indeed, constants, `3 + (4 + x)` will be rewritten to `(3 + 4) + x`. This last term will then be rewritten in an inside-out manner, with the final result being the desired `7 + x`.

We have thus used `syntaxp` to assist in specifying a strategy for simplifying sums involving constants. `Fold-consts-in-+` merely places the constants into a

position in which ACL2 can finish the job using pre-existing abilities — in this case evaluation of constant sums. Without such a rule we would have had to write a rule such as:

```
RULE: crock
REWRITE 3 + (4 + x) TO 7 + x
```

for each combination of constants encountered in the proof. Without `syntaxp` the necessity for rules such as `crock` would make ACL2 much more tedious to use.

Although the example presented here uses a very simple syntactic test in the syntaxp hypothesis, this need not be the case in general. There are ACL2 functions, not presented here, to deconstruct a term and examine the resulting pieces. Although rare, quite sophisticated syntaxp hypotheses are possible.

Before concluding this section, we wish to emphasize an important fact about syntaxp hypotheses that is easily overlooked. As mentioned above, a syntaxp hypothesis is logically true, and is treated as such during the verification of the rule containing it. Consider the rule

```
RULE: example
GIVEN integer(x)
REWRITE f(x, y) TO g(y, x)
```

in which the `integer(x)` hypothesis is required for the rule to be correct. We would not, then, be able to prove:

```
RULE: synp-example-bad
GIVEN syntaxp(x == 0)
REWRITE f(x, y) TO g(y, x)
```

Even though we, as users, know that the syntaxp hypothesis requires `x` to be the constant `0` (which is an integer), ACL2 does not get to use this fact during the proof of `synp-example-bad`. Rather we must use:

```
RULE: synp-example-good
GIVEN syntaxp(x == 0)
      integer(x)
REWRITE f(x, y) TO g(y, x)
```

ACL2 must impose this seemingly arbitrary restriction in order to maintain logical soundness. Recall that, logically speaking, syntaxp always returns true; hence the hypothesis (integerp x) does not follow logically from the syntaxp hypothesis.

3 Meta Functions

In the previous section we discussed certain aspects of the process of normalizing sums, and saw how syntaxp hypotheses can be used to achieve a greater degree of control than was possible without them. They allowed us to specify a rule based upon the ability to analyze the lexical structure of an ACL2 expression.

In this section we present facilities not only for examining a term, but also for constructing a new term, via so-called *meta rules*. These were first implemented in Nqthm, ACL2's predecessor; see [1], and we refer the reader to that paper, or the "Essay on Correctness of Meta Reasoning" in the ACL2 source code for a careful description. In a nutshell, meta rules install user-defined simplification code into the rewriter, where the user's proof obligation for those rules guarantees that each application of that code returns a term provably equal to its input. Here, we review the basics of meta rules before describing their extension in Section 4. More details may also be found in the extensive documentation distributed with ACL2, specifically within the topic "meta".

Consider the following example: arrange that for an equality between two sums, cancel any addends that are common to both sides of the equality. For instance, `x + 3*y + z == a + b + y` should be simplified to:

```
x + 2*y + z == a + b.
```

If one knew ahead of time the maximum number of addends that could appear in a sum, one could write (a large number of) rules to handle all the potential permutations in which common addends could appear; but this will not work in general and is potentially expensive in terms both of the user's labor to develop the set of rules and of ACL2's labor in sorting through such a number of rules, any particular one of which is unlikely to be needed.

Instead, we will use a *meta function*. A meta function is a custom piece of code that transforms certain terms into equivalent ones. When this transformation is proved to be correct via a *meta rule*, the meta function will be used to extend the operations of ACL2's simplifier.

Here is pseudo-code for our meta function, which cancels common summands from both sides of an equality.

```
FUNCTION: cancel-plus-equal(term)
1  if (fn-symb(term) == EQUAL
2      && fn-symb(arg1(term)) == +
3      && fn-symb(arg2(term)) == +) then
4     {lhs = sum-fringe(arg1(term))
5      rhs = sum-fringe(arg2(term))
6      int = intersect(lhs, rhs)
7      if non-empty(int) then
8         make-term(sum-tree(diff(lhs, int)),
9                   <,
10                  sum-tree(diff(rhs, int)))
11        else term}
12    else term
```

And here is the associated meta rule.

```
RULE: cancel-plus-equal-correct
META-REWRITE term TO cancel-plus-equal(term).
```

Unlike the `syntaxp` example, which merely performed a simple textual test on a term, `cancel-plus-equal-correct` takes a term and constructs an equivalent one under programmatic control.

We now examine its action on:

```
3 + f(x) + g(x, y) == x + g(x, y) + h(y, x).
```

The test of the if expression, lines 1–3, ask whether `term` is an equality between two sums. If `term` were not, `cancel-plus-equal` would return it unchanged — line 12. By returning a term unchanged, a meta function signals lack of applicability, i.e., failure. But since in the present case `term` is bound to `3 + f(x) + g(x, y) == x + g(x, y) + h(y, x)` and so is such an equality, ACL2 will execute the true-branch of the if expression in lines 4–11. Lines 4 and 5 assign to the variables `lhs` and `rhs` lists of the addends of the left-hand side and right-hand side respectively. In line 6, the intersection of these two lists is assigned to `int`. If this intersection is empty, ACL2 evaluates line 11 and returns `term` unchanged, signaling lack of applicability. Since in our case the addend `g(x, y)` is common to both sides of the equality, `int` is non-empty and so ACL2 constructs two new sums and a new equality in lines 8–10:

```
3 + f(x) == x + h(y, x).
```

The addends of these sums are the (bag-wise) difference of the two lists of addends, `lhs` and `rhs` — `{3, f(x), g(x, y)}` and `{x, g(x, y), h(y, x)}`, with the list `int` — `{g(x, y)}`.

Thus, meta rules allow one to write a custom simplifier for entire classes of terms, rather than having to write rules for a myriad of subclasses. We are able to do so because an ACL2 term is merely a structure consisting of a function symbol and the function's arguments, and we can deconstruct, examine, and reconstruct such structures using ACL2 functions.

We now briefly examine the "meaning" of a meta rule and touch upon how to prove its correctness. A meta rule not only has an associated meta function, but also has an associated evaluator that operates in an environment. An evaluator is a function that can evaluate terms by first looking in the environment for the values of any variables present in the term and then evaluating the resulting ground term. The meta rule then states that, using this evaluator, the evaluation of a manipulated term in the environment is equal to the evaluation of the original term in the same environment.

4 Extended Meta-functions

Sometime prior to ACL2 Version 2.6, one of the authors of this paper became dissatisfied with some of the limitations inherent in meta functions. In particular, he wanted to write a rule similar to `cancel-plus-equal-correct` that would cancel like factors from either side of an inequality, but was unable to do so. The difficulty stemmed from the fact that within ACL2's logic, as opposed to standard mathematics, complex numbers are linearly ordered using the dictionary

order on their real and imaginary parts respectively. Thus, for example, `0 < i` and `i < 1`. It is therefore *not* true that for numbers `x`, `y`, and `z`:

```
    0 < x
==> x*y < x*z == y < z.
```

For a counterexample, let `x` and `y` be `i`, and `z` be `0`. In contrast with a high school mathematics exam, within ACL2, one can perform such a simplification only if one also knows that `x` is rational[4]. The correct theorem in ACL2 is

```
    rational(x) &&
    0 < x
==> x*y < x*z == y < z.
```

In this section, we describe an extension to meta functions that allows us to perform such simplifications. This extension will allow us to gather information, for heuristic purposes only, that would not be otherwise available.

A "plain" meta function takes one argument — the term under consideration. This is what we saw in the previous section. An "extended" meta function takes two additional arguments, `mfc`[5] and `state`. These extra arguments give one access to functions that can be used for heuristic purposes, with names of the form `mfc-xxx`. These functions allow one to access and examine ACL2's internal data structures as well as giving one the ability to call a couple of the major functions of ACL2's rewriter.

We will see below how to make use of the following function.

```
FUNCTION: provably-pos-rat(x, mfc, state)
  mfc-rw(make-term(make-term(RATIONAL, x),
                  &&,
                  make-term(0, <, x))
         t t mfc state)
```

It asks whether ACL2 can rewrite an expression of the form `rational(x) && 0 < x` to true. We wish to emphasize here that the ability to construct and examine ACL2 terms within ACL2's logic is fundamental to such capabilities.

Here is pseudocode for our meta function, which cancels a common positive rational factor (if any) from both sides of an inequality.

```
FUNCTION: cancel-times-<(term, mfc, state)
1  if (fn-symb(term) == <
2      && fn-symb(arg1(term)) == *
3      && fn-symb(arg2(term)) == *) then
4     {lhs = product-fringe (arg1(term))
5      rhs = product-fringe (arg2(term))
```

[4] There are no irrational numbers in ACL2.

[5] Mfc stands for "Meta Function Context." The meta function context is a large and complex data structure that contains the current dynamic environment of ACL2's rewriter.

```
6       int = intersect(lhs, rhs)
7       pos-rat = find-pos-rat(int, mfc, state)
8       if non-empty(pos-rat) then
9          make-term(IF,
10                   make-term(make-term(RATIONAL, pos-rat),
11                             &&
12                             make-term(0, <, pos-rat)),
13                   make-term(product-tree(diff(lhs, pos-rat)),
14                             <,
15                             product-tree(diff(rhs, pos-rat))),
16                   term)
17         else term}
18      else term)
```

And here is the associated meta rule.

```
RULE: cancel-times-<-correct
META-REWRITE term TO cancel-times-<(term).
```

The function above is similar to `cancel-plus-equal`, but with three distinctions. First, in lines 2 and 3 `cancel-times-<` tests for the presence of products rather than sums and in lines 13 and 15 produces new products rather than sums. Second, line 7 is new. `Find-pos-rat` takes three arguments — `int` (the list of common factors), `mfc`, and `state`. `Find-pos-rat` steps through the elements of `int`, searching for one for which `provably-pos-rat(element, mfc, state)` returns true. If it is able to find one, `find-pos-rat` returns a list containing that positive, rational factor. If it is unable to find one, it returns the empty list.

Third, in lines 9–16 `cancel-times-<` constructs a more complex return value than just a simple equality between two sums or products. We must do so because, although we as users know that if `pos-rat` is non-empty it must contain a positive rational, ACL2 does not know this logically. Just as any information gathered by a syntaxp hypothesis cannot be used during verification of the rule with that hypothesis, so any information gathered by an `mfc-xxx` function is not available to ACL2 during a meta rules verification. ACL2 has no knowledge about the `mfc-xxx` functions, other than that they are functions.

We now examine the action of this rule in more detail, using

```
3 * f(x) * g(x, y) < x * g(x, y) * h(y, x)
```

as our example. We assume that in the present context, `g(x, y)` is provably a positive rational. Things will proceed much as in the previous example and, as described immediately above, `find-pos-rat` will return a list containing the single term `g(x, y)`, and this value will be assigned to the variable `pos-rat`. The test in line 8 is therefore true, and ACL2 will evaluate lines 9–16. The result is

```
if (rational(g(x, y)) && 0 < g(x, y))
   then 3 * f(x) < x * h(y, x)
   else 3 * f(x) * g(x, y) < x * g(x, y) * h(y, x).
```

During subsequent simplification of this expression, ACL2 will first rewrite the test of the `if` in order to determine which branch to use. Since (by construction) the test will rewrite to true, ACL2 will rewrite only the "then" sub-term, leading to the desired final result:

```
3 * f(x) < x * h(y, x).
```

Although this "extra" rewriting of the if expression's test might seem to be a source of inefficiency, in practice we have not found this to be true. Failure is the norm and the vast majority of the time any particular rule does not apply to the current term. Thus, only a small percentage of the work is ever duplicated, and this only when progress is (supposedly) being made.

5 Bind-Free

The careful reader may have noticed that the meta rule, `cancel-plus-equal-correct`, presented in Section 3 would not actually simplify the first, motivating, example — the addends `3 * y` and `y` are not equal, and so would not be found by merely taking the intersection of the two sets of addends. While this could be fixed by using something more sophisticated than `intersection` to determine what to subtract from both sides, we instead present a solution using a `bind-free` hypothesis.

Bind-free hypotheses grew out of a discussion between several of this paper's authors dissatisfied with the difficulty of proving simple meta rules correct. In general, this proof burden is equivalent to proving the total correctness of a piece of software. Although theoretically meta rules, plain or extended, are much more powerful than a rule using bind-free, this extra power is rarely needed. Giving up this extra power, when it is not needed, can make it much easier to write and verify the appropriate rules. This exchange, in turn, encourages one to focus upon the larger picture by proving the most general rules possible and thereby helps one to avoid getting lost in the details.

A bind-free hypothesis is similar to a syntaxp hypothesis in that its treatment when verifying the rule in which it appears differs from its treatment when that rule is being applied to a term during rewriting. Both bind-free and syntaxp hypotheses are treated as being logically true during verification of a rule, and both are evaluated during the rules application. As before this differing treatment is sound, and for the same reasons.[6]

A bind-free hypothesis differs from a syntaxp hypothesis as follows. A syntaxp hypothesis evaluates to true or false, signaling success or failure. A bind-free hypothesis either evaluates to the empty list or signals success by returning a list of pairs binding the free variables of the rule, as illustrated by the following example.

[6] ACL2 actually implements both bind-free and syntaxp using a single primitive, synp, an implementation detail that is beyond the scope of this paper.

```
FUNCTION: find-matching-addends (lhs rhs)
1  if (fn-symb(lhs) == +
2      && fn-symb(rhs) == +) then
3      {common-addends = find-common(sum-fringe(lhs),
4                                    sum-fringe(rhs))
5       if common-addends then
6          list(pair(x, common-addends))
7          else empty-list}
8      else empty-list

RULE: simplify-equality-of-sums
GIVEN rational(rhs)
      rational(lhs)
      bind-free(find-matching-addends(lhs, rhs))
REWRITE lhs == rhs TO lhs - x == rhs - x.
```

Note that it is the job of other rules, not shown here, to simplify the resulting differences.

Briefly, this rule cancels any common addends by adding their inverse to both sides of the equality. There are two things to note about this rule. First, note that the variable `x` does not appear in the left-hand side of the concluding equality of `simplify-equality-of-sums`. It is, therefore, a *free variable*. (As briefly described in the Introduction, ACL2 matches the left-hand side of a rule's concluding equality with the term currently being rewritten, binding any variables to their matching sub-terms. Since `x` does not appear in the left-hand side of `simplify-equality-of-sums`'s conclusion, it is unbound or *free*. We will make this more explicit shortly.) ACL2 has several automatic mechanisms for choosing an instantiation of such variables, which we do not discuss here. Rather, we describe how we use `bind-free` to programmatically determine the appropriate binding.

Second, the correctness of this rule does not depend upon the value that we subtract from both sides. We are free to pick this value however we want.

How is this rule applied to the following equality?

```
x + 3*y + z == a + b + y
```

As hinted in the Introduction, when ACL2 attempts to apply the rule `simplify-equality-of-sums` to the term under discussion, it first forms a substitution that instantiates the left-hand side of the rule's concluding equality so that it is identical to the target term. This substitution has the following value in our example.

```
((lhs == x + 3*y + z)
 (rhs == a + b + y))
```

ACL2 then attempts to relieve the hypotheses in the order they were given. Here, the first two hypotheses are regular ones, to be relieved by standard rewriting.

Let us assume that in the current context these hypotheses rewrite to true; we examine the final, bind-free, hypothesis.

ACL2 evaluates `find-matching-addends(lhs, rhs)` in an environment in which `lhs` and `rhs` are instantiated as determined by the substitution. In this case we evaluate

```
find-matching-addends(x + 3*y + z, a + b + y).
```

The test of the `if` expression (lines 1 and 2 above) asks whether `lhs` and `rhs` are sums. If they weren't the expression would evaluate to the empty list in line 8, signaling failure or lack of applicability. Since they are sums, ACL2 evaluates the true branch of `simplify-equality-of-sums` in lines 3 – 7. Lines 3 and 4 assign to the variable `common-addends` a list of addends common to both `lhs` and `rhs`. `Find-common` is a much more complex function than `intersection` and examines the addends in a more intelligent manner. We do not further describe this than to say that in the present case, `find-common` returns `y`, the "matching" part of `lhs` and `rhs`. The returned value of `find-matching-addends` is, therefore, `list(pair(x, y))`, informally written as `(x == y)`, and this is then used to extend the substitution:

```
((lhs == x + 3*y + z)
 (rhs == a + b + y)
 (x == y)).
```

This is used to substitute back into the `TO` side of `simplify-equality-of-sums`'s concluding rewrite, yielding the result:

```
x + 3*y + z - y == a + b + y - y.
```

Again, we have preemptively eliminated the need for a large collection of similar rules with one rule.

This rule both was able to search for matching addends in a more sophisticated manner than in `cancel-plus-equal-correct` and was easier to prove. The authors of a meta rule might be reluctant to use such a complex search method, because it could greatly complicate the proof of correctness and the simpler method was usually "good enough." A well-constructed bind-free rule, however, is often trivial to prove.

6 Extended Syntaxp

We now return to syntaxp hypotheses, but in an extended form. Just as meta rules come in two flavors, so do syntaxp hypotheses. In this section, we describe two more `mfc-xxx` functions and show how they can be used with extended syntaxp hypotheses.

(Recall that a syntaxp hypothesis is treated as being logically true when it is one of the hypotheses of the rule being proven correct. It is only during a rule's use, when the hypothesis must be relieved, that ACL2 will execute these functions. As before, any information gathered is of heuristic use only — it cannot be used to justify the correctness of a rule.)

– `mfc-clause(mfc)`: returns the current goal being proved. From the distributed `finite-set-theory` books[7] we take the following example:

```
function: rewriting-conc-lit(term, mfc, state)
subterm-of(term, last(mfc-clause(mfc)))
```

This function asks whether the term now being rewritten is the conclusion of the current goal. It has been found useful for certain expensive rules to act only upon the conclusion of a goal. The heuristic thought behind this is that often the hypotheses of a goal merely set forth the conditions under which the conclusion is true. It is therefore reasonable to expend more effort rewriting a conclusion than a hypothesis. See the `finite-set-theory/osets` books distributed with ACL2 for examples of this.

– `mfc-ancestors(mfc)`: returns the current list of the negations of the backchaining hypotheses being pursued. The only use we envision for this function is in:

```
function: rewriting-goal-literal(term, mfc, state)
is-empty-list(mfc-ancestors(mfc))
```

(Note that `term` and `state` are being ignored here.) This function asks whether we are rewriting a term from the current goal — as opposed to rewriting a hypothesis from a rewrite (or other) rule. This has been found useful in such rules as the following.

```
rule: floor-positive
given: syntaxp(rewriting-goal-literal(x, mfc, state))
       rational(x)
       rational(y)
rewrite: 0 < floor(x, y)
to: (0 < y && y <= x) || (y < 0 && x <= y)
```

By using `rewriting-goal-literal` we avoid the expense of inducing a case-split for the two disjuncts while backchaining to relieve a rule's hypotheses (when it is unlikely to do any good).

7 Extended Bind-Free

Bind-free hypotheses also come in an extended form. In this section we illustrate such hypotheses by presenting a rule for simplifying terms of the form `integer(<sum>)` where `<sum>` is a sum. For example, if we can show that `y` is an integer, we would like to simplify `integer(x + y + z)` to `integer(x + z)`.

```
FUNCTION: reduce-integer-+-fn(sum, mfc, state)
1  if fn-symb(sum) != + then
2     empty-list
```

[7] *Books* are ACL2 input files that can be run through ACL2's *certification* process.

```
3     else if provably-integer(arg1(sum), mfc, state) then
4          list(pair(int, arg1(sum)))
5     else if fn-symb(arg2(sum)) == + then
6          reduce-integer-+-fn(arg2(sum), mfc, state)
7     else if provably-integer(arg2(sum), mfc, state) then
8          list(pair(int, arg2(sum)))
9     else empty-list

RULE: reduce-integer-+
GIVEN bind-free(reduce-integer-+-fn(sum, mfc, state))
      integer(int)
REWRITE integer(sum) TO integer(sum - int)
```

We emphasize here that although we (as users) know that the second hypothesis, `integer(int)` must be true by the way that we selected `int`, this information is not available to ACL2. ACL2 must rederive this fact for its own use. Again, although this might seem a source of inefficiency, in practice we have not found this to be true.

We now consider our example, `integer(x + y + z)`, where `y` is provably an integer, and describe the action of `reduce-integer-+-fn` on this term. First note that addition is actually a binary operation in ACL2; we are really looking at the term `integer(x + (y + z))`. `Reduce-integer-+-fn` recurs on the addends so since `y`, by assumption, is provably an integer, `reduce-integer-+-fn` returns `list(pair(int, y))`. The right-hand side of the concluding equality of `reduce-integer-+` is therefore rewritten to the appropriate instance of the term `integer((x + y + z) - y)`, which will be simplified by other rules to yield our desired result.

8 Conclusion

In this paper we have described three facilities afforded by ACL2 for varying levels of meta level control and reasoning. The weakest of these, syntaxp hypotheses, allow one to control the behavior of ordinary rewrite rules by restricting their operations based upon the syntactic form of their instantiated variables. A quick search reveals that there are over 800 uses of syntaxp hypotheses among more than 150 of the proof scripts distributed with ACL2. Bind-free hypotheses, which allow one to programmatically select a binding for free variables, are slightly more powerful and there are more than 50 uses of this relatively new facility in approximately 15 scripts. Finally, there are about 25 uses of meta rules in 12 scripts[8].

[8] It seems likely that all but a couple of these meta rules could, instead, be made bind-free rules. For example, the meta rule `cancel-plus-equal-correct` could be replaced with the more general bind-free rule `simplify-equality-of-sums`. That this is not the case is due to the fact that bind-free rules were not available at the time of many of these meta rules' creation.

The initial designs of these facilities sprang from users' frustrations with the effort required to carry out certain proofs which required one to continually point out to ACL2 simple and seemingly "obvious" steps. The initial implementations were then tested, generalized, and tested again. They were not placed into the main development branch of ACL2's source code until all were satisfied that they were correctly and efficiently implemented, as easy to use as possible, and sufficiently general to be broadly useful.

As we have seen, these meta level facilities allow one to specify solutions to entire classes of problems, avoiding the need for a myriad of rules in their stead. This allows one to concentrate on the structure of the desired proof, and to leave many of the details to ACL2. See [7] for an example of this in the domain of bags, or multisets.

Acknowledgements

We thank the referees, who provided significant useful feedback (after obvious careful reading). We also thank the ACL2 community for useful discussions.

References

1. R. S. Boyer and J Moore. Metafunctions: proving them correct and using them efficiently as proof procedures. The Correctness Problem in Computer Science, R. S. Boyer and J Moore, editors. Accedemic Press, 1981.
2. M. J. C. Gordon and T. F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, 1993.
3. J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique.
4. M. Kaufmann, P. Manolios, and J Moore. Editors. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, 2000.
5. M. Kaufmann, P. Manolios, and J Moore. Editors. Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, 2000.
6. M. Kaufmann and J Moore. ACL2: An Industrial Strength Version of Nqthm. Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96), pp. 23-34, IEEE Computer Society Press, June 1996.
7. Eric Smith, Serita Nelesen, David Greve, Matthew Wilding, and Raymond Richards. An ACL2 Library for Bags (MultiSets). Fifth International ACL2 workshop, 2004. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/index.html`
8. Markus Wenzel. Isar — a Generic Interpretive Approach to Readable Formal Proof Documents. Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, LNCS 1690, Springer, 1999.

Reasoning About Java Programs with Aliasing and Frame Conditions*

Claude Marché and Christine Paulin-Mohring

PCRI: LRI (Université Paris-Sud & CNRS UMR 8623), INRIA Futurs,
Bât. 490, Université Paris-Sud, 91405 Orsay Cedex, France

Abstract. Several tools exist for reasoning about JAVA programs annotated with JML specifications. A main issue is to deal with possible *aliasing* between objects and to handle correctly the *frame conditions* limiting the part of memory that a method is allowed to modify. Tools designed for automatic use (like ESC/JAVA) are not complete and even not necessarily correct. On the other side, tools which offer a full modeling of the program require a heavy user interaction for discharging proof obligations. In this paper, we present the modeling of JAVA programs used in the KRAKATOA tool, which generates proof obligations expressed in a logic language suitable for both automatic and interactive reasoning. Using the SIMPLIFY automatic theorem prover, we are able to establish automatically more properties than static analysis tools, with a method which is guaranteed to be sound, assuming only the correctness of our logical interpretation of programs and specifications.

1 Introduction

KRAKATOA [16] is a prototype tool for verifying that a JAVA program meets its JML specification (JML stands for Java Modeling Language [14, 15]). It is built on top of the WHY tool [11], which generates proof obligations from annotated programs written in a basic ad-hoc programming language with higher-order functions, references, exceptions, and a simple modular specification language [10]. These proof obligations can be generated for various interactive proof assistants and automatic theorem provers.

KRAKATOA expresses the operational semantics of a JAVA program by producing a translation into the WHY programming language as well as translating the JML specification into logical assertions. KRAKATOA needs also to provide a theory corresponding to the program which expresses the representation of the memory, and the dynamic typing information. The memory modeling of the first version of KRAKATOA was built on a unique heap where the objects and arrays were stored, and the theory was only generated for the COQ proof assistant [21].

* Research partly supported by the EU IST project IST-2000-26328-VERIFICARD (`www.verificard.org`), the GECCOO project of the "ACI Sécurité Informatique" (`geccoo.lri.fr`) and the EU coordinated action TYPES (`www.cs.chalmers.se/Cs/Research/Logic/Types/`).

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 179–194, 2005.

The tool was successfully used for proving small examples like Dijkstra's Dutch Flag algorithm or basic properties of a method in a JavaCard applet provided by the Schlumberger company for the IST VerifiCard project [8, 13]. The modular proof architecture appeared well-suited for dealing with large programs, however the manual work for proving proof obligations was complicated due to the lack of automation, and the too naive memory representation which was not taking enough static typing information into account.

To cope with this problem, we changed the modeling and adopted a more local representation of the memory. This alternative approach, already present in early work on general pointer programs by Burstall [7] is emphasized by Bornat [4]. It amounts to associate to each structure field a map from adresses to value, access or modification of the field of some structure being just interpreted as the corresponding access or update of the map at the index corresponding to the address of the structure. This approach works perfectly well with Java object fields, because the corresponding cell can only be accessed using the field name, but we also had to extend the approach to Java arrays, and also to support new memory allocation. This was not a trivial task because the Krakatoa tool has to perform an accurate static analysis of the program in order to deal properly with frame conditions (JML `modifiable` clauses, specifying the only part of the memory that can be changed by a method). Another extension was to provide a first-order theory for the logical aspects of the programs in order to use automatic provers such as Simplify [19] for solving proof obligations. Building a first-order theory for Java programs was also non-trivial and we used our higher-order Coq modeling for validating the axioms in order to avoid building an inconsistent theory that will trivially solve all proof obligations. The new version of Krakatoa together with Simplify can automatically check simple properties of programs (no null object dereferencing, no out-of-bounds array access, etc.): it validates proof obligations or provides counter-examples. Unlike ESC/Java which does not properly check frame properties leading to accept programs which are wrong, the Krakatoa approach never gives wrong positive answers.

Section 2 gives an overview of the languages and notations, mainly JML and the specification language of the Why tool. Section 3 explains our modeling of memory states and describes the background theory needed to solve proof obligations. Section 4 explains how Java programs are interpreted in our model. Section 5 studies in more details how the frame conditions are handled. Section 6 illustrates our method on some examples. We conclude in Sect. 7, with comparison to related works.

2 Preliminaries

2.1 Considered Fragment of JML

In JML, Java programs can be annotated using a special class of comments. Logical formulas are written as Java boolean expressions using only *pure* methods (i.e. without side effects). Furthermore some special operators such as `\forall`,

```
class Purse {
    int balance; //@ invariant balance >= 0;

    /*@ normal_behavior
      @  requires s >= 0;
      @  modifiable balance;
      @  ensures balance == \old(balance)+s; */
    public void credit(int s) { balance += s; }

    /*@ behavior
      @  requires s >= 0;
      @  modifiable balance;
      @  ensures s <= \old(balance) && balance == \old(balance) - s;
      @  signals (NoCreditException)
      @         s > balance && balance == \old(balance); */
    public void withdraw(int s) throws NoCreditException {
        if (balance >= s) { balance -= s; }
        else { throw new NoCreditException(); }
    }
}
```

Fig. 1. JML specification of a simplified electronic purse

`\exists` are introduced. In a post-condition, the construction `\result` refers to method's returned value, and `\old`(e) is the value of expression e before the execution of the method.

In this paper, we focus on the most important features of JML: declarations of class invariants and method annotations describing their functional behavior: a `requires` (resp. `ensures`) clause gives the pre-condition (resp. the post-condition), and a `modifiable` clause, also called *frame condition*, specifies the set of memory locations where the method can write. A `signals` clause similar to `ensures` specifies which exception can be raised and which properties are true in that case. This is illustrated on Figure 1 which introduces a simple class `Purse` with a field `balance` which should be non-negative (invariant), a method `credit` (resp. `withdraw`) which adds (resp. removes) money to the purse.

Krakatoa also supports other essential JML constructs such as loop invariants. Regarding exceptions, it is noticeable that the Krakatoa approach, whatever the JML specification is, inserts pre-conditions to access operations which prevent to raise an exception in the class `RunTimeException` (mainly `NullPointerException` or `ArrayIndexOutOfBounds`). Also, we interpret `byte`, `short`, etc. into unbounded integer arithmetic (arithmetic overflow could be forbidden by insertion of suitable preconditions on integer operations [16]).

2.2 Interpretation into the Why Tool

Why's core language includes higher-order functions, references and exceptions but rejects any variable aliasing. Programs can be annotated using pre, post-conditions, loop invariants and intermediate assertions. Logical formulas are

written in a typed (sorts, possibly parametric) first-order logic, with built-in equality and integer arithmetic. WHY performs an *effect analysis* (see below) on programs, a classical weakest-precondition computation, and generates proof obligations for various provers.

WHY has a primitive notion of exceptions which is integrated in its weakest pre-condition calculus. It is used for the interpretation of `break` and `continue` statements or the JAVA exception mechanism and JML exceptional behavior (see [16]). WHY has a modular approach: new logic functions or predicate symbols can be introduced and freely used in the specification part, and sub-programs can also be introduced abstractly, giving just a full specification: type of input variables and result, pre- and post-conditions, as well as its *effects*, that is giving which of the global references can be read and/or written by the sub-program.

In order to translate a JAVA program annotated with JML into WHY, one needs to proceed the following way:

1. Find an appropriate modeling of JAVA memory states using global WHY variables which will never be aliased ;
2. Translate JAVA constructs into WHY statements, with assignments over the global variables defined before ;
3. Translate JML formulas into WHY predicates, over those variables.

Our JAVA modeling contains a generic part which is the same for all JAVA programs and a specific part which depends on the particular class hierarchy. The generic part introduces a global variable `alloc` which keeps track of allocated objects, it defines the set of values, memory segments (mapping addresses to values), JAVA types, and operations such as access or update of memory as well as logical relations like to be an instance of a class (`instanceof`), or to preserve part of the memory (`modifiable`). These notions will be formally introduced in the next sections.

The specific part introduces constants for the different classes (`Purse` in the example) and global variables for the fields (`balance` in the example), it also defines a predicate for each class invariant.

Because JAVA methods in a class can be mutually recursive, we first give a specification of all the methods as abstract WHY sub-programs with specifications. This abstract view is used to interpret method calls, and suffices to ensure partial correctness of programs.

The general form of a WHY program specification f is the following :

parameter f: x_1:$\mathrm{type}_1 \rightarrow \cdots \rightarrow \mathrm{x}_n$:$\mathrm{type}_n \rightarrow \{$ *Precondition* $\}$ *return_type*
reads *input_vars* **writes** *output_vars*
$\{$ *Postcondition* $\mid$ $exception_1 \Rightarrow condition_1 \ \ldots \ exception_n \Rightarrow condition_n$ $\}$

Figure 2 shows the WHY abstract declaration of the sub-program corresponding to the `Purse.credit` method of Figure 1. It has two arguments (the object *this* and the integer parameter s) which are supposed to satisfy the pre-condition (first formula between curly braces). There is no result (output type *unit*), it

parameter Purse_credit : $this$: value $\rightarrow$ s : int $\rightarrow$
{ $s \geq 0 \wedge this \neq$ Null $\wedge$ (instanceof $alloc$ $this$ (ClassType Purse))
$\wedge$(Purse_invariant $balance$ $this$)}
unit **reads** $balance, alloc$ **writes** $balance$
{ (access $balance$ $this$) = (access $balance$@ $this$) + $s \wedge$ (Purse_invariant $balance$ $this$)
$\wedge$(modifiable $alloc$@ $balance$@ $balance$ (value_loc $this$)) }

Fig. 2. WHY interpretation of `Purse.credit`

accesses two global variables (*balance* and *alloc*) and writes one (*balance*). The post-condition (second pair of curly braces) uses the notations *balance*@ and *alloc*@ to denote the value of these variables before the method application.

The WHY language has a formally defined semantics [10], and the fact that the generated proof obligations are sufficient conditions for the program to meet its specification is furthermore guaranteed by a *validation* which is built for each function and can be automatically checked. However, the main source of error in our method could be that our translation of JAVA programs or JML specification does not respect the JAVA/JML semantics. For these reasons, it is important for these interpretations to be clearly stated. This is the purpose of the following sections.

3 Modeling Java Memory States

We have to represent states of JAVA memory by a finite set of WHY variables, and describe the state transitions corresponding to JAVA statements as modifications of those variables.

For local variables of constructors or methods, since such variables are allocated in JAVA memory stack, and cannot be aliased to another local variable or a cell of memory heap, it is sound to represent them as local variables in WHY intermediate language.

3.1 Modeling the Heap

JAVA values are either direct values (integers, booleans or floats) or references to objects or arrays which are represented as adresses allocated in the heap. As mentioned in Sect. 1, the first version of KRAKATOA described in [16] used a naive method considering JAVA memory heap as a single large array mapping addresses to values, but this modeling is very low-level, and proving properties with it amounts to reason all the time on whether two addresses are aliased or not: for example, if a field x is modified, one should expect that it is known for free that any different field y is unchanged ; or when an array cell is modified, the length of the array is not. This is why we adapted Burstall's approach for pointer programs with structures to JAVA programs.

The set of variables representing a state of Java memory heap is displayed on Fig. 3. All these variables can be seen as maps indexed by addresses $a_1, a_2, \ldots$ belonging to some abstract data type addr.

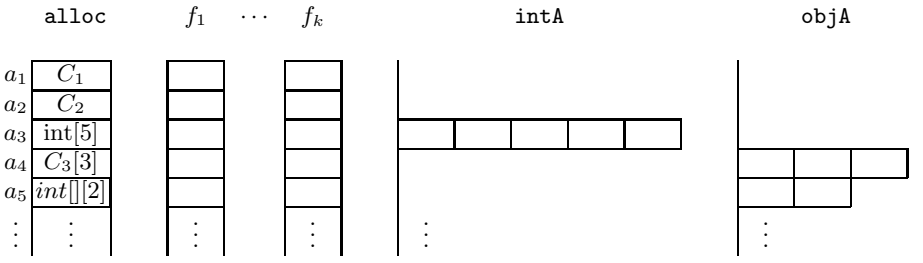


Fig. 3. Modeling of Java memory heap

The variable `alloc` on the left of the figure, is an *allocation store* (type store) which tells for each address whether it is allocated, and if yes what is the type of the structure at this address: an object of some class C or an array of some length l of values of some type t. This variable will be accessed when length of an array is sought and for dynamic typing (`instanceof` and casts), and modified only by `new` statements. The variables $f_1, f_2, \ldots$ represent dynamic fields of objects, in a very similar way as Burstall.

The variables `intA` and `objA` represent the memory locations where arrays of integers and references respectively are allocated (for simplicity, we only consider the basic type `int` here, but in practice booleans and floats are also handled, with variables `boolA` and `floatA`). When an array arr of integers of size l is allocated at address a, then $alloc(a)$ will be equal to $\texttt{int}[l]$ and for all integer $0 \leq i < l$, $\texttt{intA}(a, i)$ will be an integer corresponding to $arr[i]$. An array of objects will be allocated similarly in the variable `objA`. Notice that the array `objA` cannot be split further because it is possible for any two arrays of objects to be aliased (think of arguments of `System.arraycopy` method of Java API).

3.2 First-Order Modeling of Values, Classes and Memories

The next step is to design a first-order theory, introducing function and predicate symbols and axioms for them, to model Java execution in terms of formulas over the variables introduced in the previous section.

A very useful consequence of our splitting of the heap is that it is statically known whether the contents of some memory cell is an integer or a reference; hence in the logical modeling, instead of a unique sort for representing any memory cell value, we use primitive integers of the logic for values of arithmetic expressions plus a sort value for values corresponding to references (objects or arrays). This sort value is equipped with a function Ref : addr → value which builds an object from an address, and a logical constant Null of type value for representing Java's `null` value.

We introduce a sort javaType for representing Java types of references. With classId being the type of class names of the given Java program, javaType is constructed by:

$\mathsf{ClassType} : \mathsf{classId} \to \mathsf{javaType}$ // for class C
$\mathsf{ArrIntType} : \mathsf{javaType}$ // for array int[]
$\mathsf{ArrayType} : \mathsf{javaType} \to \mathsf{javaType}$ // for array t[], with t a reference type

The `alloc` variable of type `store` corresponds to a finite partial map which contains the currently allocated values with type information. In practice we shall use the following functions to access type information.

$\mathsf{fresh} : \mathsf{store} \to \mathsf{value} \to \mathsf{prop}$
$\mathsf{typeof} : \mathsf{store} \to \mathsf{value} \to \mathsf{javaType} \to \mathsf{prop}$
$\mathsf{arraylength} : \mathsf{store} \to \mathsf{value} \to \mathsf{int}$

fresh and typeof corresponds respectively to JML's `\fresh` and `\typeof`; arraylength corresponds to `.length` in Java or JML. Derived from the typeof relations, we shall introduce logical interpretation of Java predicate instanceof and JML function `\elemtype`:

$\mathsf{instanceof} : \mathsf{store} \to \mathsf{value} \to \mathsf{javaType} \to \mathsf{prop}$
$\mathsf{array\_elemtype} : \mathsf{store} \to \mathsf{value} \to \mathsf{javaType}$

Useful properties on these functions are

$$\forall\ s : \mathsf{store}, v : \mathsf{value},\ 0 \leq (\mathsf{arraylength}\ s\ v),$$
$$\forall\ s : \mathsf{store}, t : \mathsf{javaType}, v : \mathsf{value},\ (\mathsf{typeof}\ s\ v\ t) \to (\mathsf{instanceof}\ s\ v\ t)$$
$$\forall\ s : \mathsf{store}, t : \mathsf{javaType}, v : \mathsf{value},\ (\mathsf{instanceof}\ s\ v\ t) \to \neg(\mathsf{fresh}\ s\ v)$$
$$\forall\ s : \mathsf{store}, t : \mathsf{javaType}, v : \mathsf{value},$$
$$v \neq \mathsf{Null} \wedge (\mathsf{typeof}\ s\ v\ (\mathsf{ArrayType}\ t)) \to (\mathsf{array\_elemtype}\ s\ v) = t$$

And for each class C extending class D:

$$\forall\ s : \mathsf{store}, v : \mathsf{value},$$
$$(\mathsf{instanceof}\ s\ v\ (\mathsf{ClassType}\ C)) \to (\mathsf{instanceof}\ s\ v\ (\mathsf{ClassType}\ D))$$

The store will only be changed when a new object is allocated. The main information we need is that the new store contains the objects allocated in the old store. This is achieved by the introduction of a binary relation store_extends on stores with the following properties (among others):

$$\forall\ s_1, s_2 : \mathsf{store}, v : \mathsf{value},\ (\mathsf{store\_extends}\ s_1\ s_2) \wedge (\mathsf{fresh}\ s_2\ v) \to (\mathsf{fresh}\ s_1\ v)$$
$$\forall\ s_1\ s_2 : \mathsf{store}, t : \mathsf{javaType}, v : \mathsf{value},$$
$$(\mathsf{store\_extends}\ s_1\ s_2) \wedge (\mathsf{instanceof}\ s_1\ v\ t) \to (\mathsf{instanceof}\ s_2\ v\ t)$$

Krakatoa introduces one additional Why variable (f_i) for each static field of the Java program. Variable f_i will have type α memory where α is either int or value, depending of the static type declared for f_i. The basic operations on objects of type α memory are access and update:

$\mathsf{access} :\ \alpha\ \mathsf{memory} \to \mathsf{value} \to \alpha$
$\mathsf{update} :\ \alpha\ \mathsf{memory} \to \mathsf{value} \to \alpha \to \alpha\ \mathsf{memory}$

To reason with combination of access and updates, we have the classical properties of the theory of arrays:

$$\forall\ m : \alpha\ \mathsf{memory}, v : \mathsf{value}, w : \alpha,\ (\mathsf{access}\ (\mathsf{update}\ m\ v\ w)\ v) = w$$
$$\forall\ m : \alpha\ \mathsf{memory}, v_1, v_2 : \mathsf{value}, w : \alpha,$$
$$v_1 \neq v_2 \rightarrow (\mathsf{access}\ (\mathsf{update}\ m\ v_1\ w)\ v_2) = (\mathsf{access}\ m\ v_2)$$

The variables `intA` and `objA` have type int arraymem and value arraymem respectively. These maps are indexed by both a value and an integer, with operations

$$\mathsf{array\_access} : \alpha\ \mathsf{arraymem} \rightarrow \mathsf{value} \rightarrow \mathsf{int} \rightarrow \alpha$$
$$\mathsf{array\_update} : \alpha\ \mathsf{arraymem} \rightarrow \mathsf{value} \rightarrow \mathsf{int} \rightarrow \alpha \rightarrow \alpha\ \mathsf{arraymem}$$

and the expected properties for combination of access and updates.

3.3 Coq Realization

We use Why's ability to translate formulas into several prover output formats, in particular for the Coq proof assistant and for the Simplify automatic theorem prover. With Coq output, we furthermore built a *realization* of all axioms of our theory. This has the very important consequence that the original first-order theory is proven consistent.

Our Coq development is structured as a functor, whose parameter is a signature representing the class structure of an arbitrary Java program, made of:

- a set of class identifiers classId;
- a distinguished class identifier ObjectClass (for the Java `Object` class);
- a partial function super associating to its class its superclass ;
- three axioms for assuming decidability of equality on classId, that ObjectClass as no superclass, and that super is a well-founded relation.

From this signature, the functor builds a module which provides a realization of the previous first-order theory, using inductive data types (for representing sorts value, javaType, etc.) and higher-order functions, intensively used in order to represent the memory types and operations, and the sets of memory locations presented further in Sect. 5. It contains 1500 lines of Coq code. We have also introduced 200 lines of specialized tactics in order to mechanize simple reasoning.

On each particular program, Krakatoa generates an instance of the signature above, thus providing a proven consistent modeling of this program.

4 Translating Java Programs

We now give the semantical interpretation of Java statements which access and/or updates the memory heap. The interpretation of complex statements in Java (sequence, if, while, exception throwing and catching, etc.) is not different from [16] so we focus here only on atomic statements of access, assignment, and memory allocation.

4.1 Analysis of Variable Effects

Once we have fixed the set of variables representing memory states, one very important step for being able to deal modularly with function calls in WHY's intermediate language, is to statically compute for any statements the variable effects, as it is shown for example in **reads** and **writes** clauses of Figure 2. This analysis of effects allows to interpret JML `modifiable` clauses, as it will be shown in Sect. 5.

Because JAVA methods in a class can be mutually recursive, the effect analysis uses an iterative process in order to compute the maximal effects of each method, starting from an empty effect. We introduce an environment Γ which associates to each method its (currently known) effects, we write $\Gamma \vdash e : R, W$ to mean that expression (or statement) e reads variables R and writes variables W assuming the methods have effects as given in Γ. We give here a few rules for computing $\Gamma \vdash e : R, W$:

$$\frac{\Gamma \vdash e : R, W}{\Gamma \vdash e.f : R \cup \{f\}, W}$$

$$\frac{e_1 \text{ has type } \texttt{int[]} \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \Gamma \vdash e_2 : R_2, W_2 \qquad \Gamma \vdash e_3 : R_3, W_3}{\Gamma \vdash e_1[e_2] = e_3 : R_1 \cup R_2 \cup R_3 \cup \{\texttt{intA}, \texttt{alloc}\}, W_1 \cup W_2 \cup W_3 \cup \{\texttt{intA}\}}$$

$$\frac{m : R, W \text{ in } \Gamma \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \cdots \qquad \Gamma \vdash e_k : R_k, W_k}{\Gamma \vdash m(e_1, \ldots, e_k) : \bigcup_i R_i \cup R, \bigcup_i W_i \cup W}$$

$$\frac{C : R, W \text{ in } \Gamma \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \cdots \qquad \Gamma \vdash e_k : R_k, W_k}{\Gamma \vdash \texttt{new } C(e_1, \ldots, e_n) : \bigcup_i R_i \cup R \cup \{\texttt{alloc}\}, \bigcup_i W_i \cup W \cup \{\texttt{alloc}\}}$$

From a given Γ, and a given method m with body e, we compute R, W such that $\Gamma \vdash e : R, W$ and update consequentely the information associated to method m in Γ until a fixpoint is reached, which happens in finite time because set of effects are bounded: the number of variables is fixed.

4.2 Memory Access

In our logical model of JAVA program, we introduced total functions in order to be able to represent any well-typed JAVA program. But we want also to detect and avoid any possible runtime exception such as access to a null pointer, or outside the bounds of an array.

Let's consider a Java access expression `v.f`, where v is a variable (the general case of any expression can be dealt by adding a temporary variable) and f a field name. The logical interpretation views f as something of type α memory where α is either int or value, depending on the static type of f. The logical interpretation of `v.f` is (access f v) using the access function introduced in our model (see Sect. 3.2). But this access will generate a runtime exception if v is the null pointer. It is necessary to produce a proof obligation $v \neq \mathsf{Null}$. One possibility could be to introduce in WHY an `access` function with the corresponding precondition. We prefer to use the possibility in WHY to associate pre

or post-condition to any expression. The JAVA `v.f` expression is consequentely translated into the WHY annotated expression: $\{\ v \neq \mathsf{Null}\ \}\ (\mathsf{access}\ f\ v)$. The precondition is omitted when v is the self reference `this`.

Similarly, an array access expression `v[i]` is interpreted as the annotated expression $\{\ v \neq \mathsf{Null} \wedge 0 \leq i < (\mathsf{arraylength}\ \mathtt{alloc}\ v)\ \}\ (\mathsf{array\_access}\ A\ v\ i)$, where $A =$ `intA` or `objA` depending on static type of the array. The precondition ensures that no null pointer access and no out-of-bounds access occur.

4.3 Memory Assignments

A field assignment `v.f = w` is interpreted as $f := \{\ v \neq \mathsf{Null}\ \}\ (\mathsf{update}\ f\ v\ w)$. Updating the field f of object v is protected by the condition that v should not be `null`, another natural condition is to check that the type of the object w is an instance of the type of the field f. However, this is a consequence of static typing and type-safety in JAVA so we do not need to add extra checking.

The situation is different for array assignement `v[i] = w`. Assume D is a subclass of C, it can be the case that v is statically an array of C, but dynamically an array of D in which case updating v with an object w in the class C is statically correct but fails at runtime. In order to avoid this error, we interpret `v[i] = w` into

$$A := \{\ v \neq \mathsf{Null} \wedge 0 \leq i < (\mathsf{arraylength}\ \mathtt{alloc}\ v) \wedge (\mathsf{instanceof}\ \mathtt{alloc}\ w\ (\mathsf{array\_elemtype}\ \mathtt{alloc}\ v))\ \}\ (\mathsf{array\_update}\ A\ v\ i\ w)$$

4.4 Memory Allocation

A JAVA object creation expression `new C`$(v_1, \ldots, v_n)$ is interpreted as

$$\mathtt{let}\ this = (\mathsf{alloc\_obj}\ C)\ \mathtt{in}\ C_{fun}(this, v_1, \ldots, v_n); this$$

where C_{fun} is the WHY function for the coresponding constructor. Unlike access and update functions, alloc_obj is a function with side-effects, specified in WHY:

parameter alloc_obj : c : classId $\rightarrow \{\}$ value **reads** `alloc` **writes** `alloc`
$\{\ result \neq \mathsf{Null} \wedge (\mathsf{fresh}\ \mathtt{alloc@}\ result) \wedge (\mathsf{typeof}\ \mathtt{alloc}\ result\ (\mathsf{ClassType}\ c)) \wedge (\mathsf{store\_extends}\ \mathtt{alloc@}\ \mathtt{alloc})\}$

Array creation `new C[l]` is interpreted as $(\mathsf{alloc\_array}\ C\ l)$ where alloc_array is specified as

parameter alloc_array : t : javaType $\rightarrow n$: int $\rightarrow \{0 \leq n\}$ value
reads `alloc` **writes** `alloc`
$\{\ result \neq \mathsf{Null} \wedge (\mathsf{fresh}\ \mathtt{alloc@}\ result) \wedge (\mathsf{typeof}\ \mathtt{alloc}\ result\ (\mathsf{ArrayType}\ t)) \wedge (\mathsf{arraylength}\ \mathtt{alloc}\ result) = n \wedge (\mathsf{store\_extends}\ \mathtt{alloc@}\ \mathtt{alloc})\}$

There is also a special variant alloc_int_array for `new int[l]`.

Notice that in order to prove JAVA programs, we do not need to know how these functions are implemented. However, in order to avoid axioms in our model,

the alloc_obj and alloc_array functions have also been implemented in Why using more primitive functional operations for computing a non allocated value in a store and creating an updated store. We have a Coq proof of correctness of these implementations (technically, this requires the addr type to be infinite, since we do not consider memory overflow).

5 Modeling Frame Conditions

JML `modifiable` clauses are essential for reasoning modularly when several methods are involved. As a toy example, let's imagine a new method in our class Purse, given Fig. 4. Proof of the post-condition needs the fact that `p1.balance` is not modified by the call to `p2.credit(100)`. Our modeling allows to prove this using the modifiable predicates in the post-condition of `credit` and the pre-condition `p1 != p2`, that forbids aliasing of `p1` and `p2`.

We model `modifiable` clauses using the predicates

$$\mathsf{modifiable}: \ \mathsf{store} \to \alpha\ \mathsf{memory} \to \alpha\ \mathsf{memory} \to \mathsf{set\_loc} \to \mathsf{prop}$$
$$\mathsf{array\_modifiable}: \ \mathsf{store} \to \alpha\ \mathsf{arraymem} \to \alpha\ \mathsf{arraymem} \to \mathsf{array\_set\_loc} \to \mathsf{prop}$$

respectively for objects locations and array locations. set_loc (resp. array_set_loc) are logic types representing sets of modifiable locations for objects (resp. for arrays).

In general, the post-condition of a method will have one modifiable predicate for each of the Why variables it modifies, as they are computed by the analysis of effects of Sect. 4.1. Splitting of the JML `modifiable` clause into modifiable predicate for each modified variable is computable automatically.

According to JML informal semantics, a `modifiable` clause with set of locations *loc* specifies that in the post-state of the considered method, every memory location which is already allocated in the pre-state and is not included in *loc* is unchanged. This formally results in the following:

$$\forall\ s : \mathsf{store}, m_1, m_2 : \alpha\ \mathsf{memory}, loc : \mathsf{set\_loc},$$
$$(\mathsf{modifiable}\ s\ m_1\ m_2\ loc) \leftrightarrow$$
$$\forall\ v : \mathsf{value},\ \neg(\mathsf{fresh}\ s\ v) \wedge (\mathsf{notin}\ v\ loc) \to (\mathsf{access}\ m_1\ v) = (\mathsf{access}\ m_2\ v)$$

```
/*@ normal_behavior
  @   requires p1 != null && p2 != null && p1 != p2;
  @   modifiable p2.balance;
  @   ensures \result == \old(p1.balance); */
public static int test(Purse p1,Purse p2) {
    p2.credit(100);
    return p1.balance;
}
```

Fig. 4. An example of reasoning on aliases and frame conditions

$$\forall\ s : \mathit{store}, m_1, m_2 : \alpha\ \mathsf{arraymem}, \mathit{loc} : \mathsf{array\_set\_loc},$$
$$(\mathsf{array\_modifiable}\ s\ m_1\ m_2\ \mathit{loc}) \leftrightarrow$$
$$\forall\ v : \mathsf{value},\ n : \mathsf{int}, \neg(\mathsf{fresh}\ s\ v) \wedge (\mathsf{array\_notin}\ v\ n\ \mathit{loc}) \rightarrow$$
$$(\mathsf{array\_access}\ m_1\ v\ n) = (\mathsf{array\_access}\ m_2\ v\ n)$$

and it remains to give axioms for the notin (resp. array_notin) functions, depending on the form of the locations specified.

The JML clause `modifiable \nothing` specifies that nothing is modified. It is interpreted with a new constant empty_loc of type set_loc (resp. array_empty_loc of type array_set_loc) with the axioms:

$$\forall v : \mathsf{value}, \quad (\mathsf{notin}\ v\ \mathsf{empty\_loc})$$
$$\forall v : \mathsf{value}, n : \mathsf{int}, \quad (\mathsf{array\_notin}\ v\ n\ \mathsf{array\_empty\_loc})$$

The JML clause `modifiable v.f` specifies that the field f of v is modified. It is interpreted with a new function value_loc of type value → set_loc with the axiom:

$$\forall v'\ v : \mathsf{value}, \quad (\mathsf{notin}\ v'\ (\mathsf{value\_loc}\ v)) \leftrightarrow v' \neq v$$

Analogously, The JML clauses `modifiable t[i]`, `modifiable t[i..j]` and `modifiable t[*]` are interpreted using functions array_loc, array_sub_loc and array_all_loc with axioms

$$\forall\ v\ t : \mathsf{value},\ n\ i : \mathsf{int}, (\mathsf{array\_notin}\ v\ n\ (\mathsf{array\_loc}\ t\ i)) \leftrightarrow (v \neq t \vee i \neq n)$$
$$\forall\ v\ t : \mathsf{value},\ n\ i\ j : \mathsf{int},$$
$$(\mathsf{array\_notin}\ v\ n\ (\mathsf{array\_sub\_loc}\ t\ i\ j)) \leftrightarrow v \neq t \vee n < i \vee n > j$$
$$\forall\ v\ t : \mathsf{value},\ n : \mathsf{int}, (\mathsf{array\_notin}\ v\ n\ (\mathsf{array\_all\_loc}\ t)) \leftrightarrow v \neq t$$

When a JML clause `modifiable l1,l2` specifies several (say two) locations l_1 and l_2, then two cases may occur: either l_1 and l_2 refer to locations represented by different variables of the memory heap representation, and in that case a conjunction of two modifiable assertions is built; or they refer to the same variable, and then the clause is interpreted using the function union_loc (resp. array_union_loc) with axioms

$$\forall v : \mathsf{value},\ l_1, l_2 : \mathsf{set\_loc},$$
$$(\mathsf{notin}\ v\ (\mathsf{union\_loc}\ l_1\ l_2)) \leftrightarrow (\mathsf{notin}\ v\ l_1) \wedge (\mathsf{notin}\ v\ l_2)$$
$$\forall v : \mathsf{value}\ n : \mathsf{int},\ l_1, l_2 : \mathsf{array\_set\_loc},$$
$$(\mathsf{array\_notin}\ v\ n\ (\mathsf{array\_union\_loc}\ l_1\ l_2))$$
$$\leftrightarrow (\mathsf{array\_notin}\ v\ n\ l_1) \wedge (\mathsf{array\_notin}\ v\ n\ l_2)$$

Notice also that if a variable is detected as written by the analysis of effects, but there is no `modifiable` location refering to it, then we have to add an assertion (modifiable $\cdots$ empty_loc) for it. Finally, the JML clause `modifiable \everything` is interpreted simply by building no modifiable predicate, for specifying no information at all.

These constructions have been implemented in our Coq realization. The sort set_loc is interpreted as the functional type value → Prop representing intentionally a set of locations. We interpret a set of locations directly as the predicate which is true for values which are not in this set of locations, such that the predicate notin can be interpreted directly without extra negation.

6 Examples

ESC/Java does not check that a method meets the frame condition written in its specification, but it assumes that this condition is fulfilled when the method is called. This is one of the major sources of unsoundness. The ChAsE [9] tool was designed for automatically checking frame conditions, but it works at a syntactic level and consequently can give incorrect diagnosis: two such examples are given [9] (see Figure 5). ChAsE accepts these programs with incorrect frame conditions, but not with the appropriate ones which are `a[i+1]` instead of `a[i]` for method `q` and `y.i` instead of `x.i` for method `p`. On the other hand, Krakatoa gives automatically the correct diagnosis for both programs. For instance for `p` with clause `modifiable y.i`, which, according to JML semantics, denotes the field i at address $this.y$ in the pre-state of the method: Krakatoa interprets this clause as (modifiable `alloc`@ i@ i (value_loc (access y@ $this$))) which is easily provable.

```
class Q {
  int i; int[] a;

 /*@ normal_behavior
   @  requires 0<=i && i+1 < a.length;
   @  modifiable i,a[i]; */
  void q() { i++; a[i]=3; }
}
```

```
class O { int i; }
class P {
  O x; O y;
 /*@ normal_behavior
   @  requires y != null;
   @  modifiable x,x.i;  */
  void p() { x=y; x.i=7; }
}
```

Fig. 5. Two programs where ChAsE gives the wrong answer

In practice, most proof obligations generated are solved automatically using Simplify, for example all obligations of the Krakatoa tutorial (a simple electronic purse, maximum of an array, etc.) and the Dijkstra's Dutch Flag program of [16]. In comparaison, using Coq with simple ad-hoc tactics, the proofs for the `Purse` (resp. `Flag`, resp. `Arrays`) programs require 20 (resp. 60, resp. 100) lines of tactics.

7 Conclusions, Related Works, and Future Work

7.1 Combining First-Order and Higher-Order Models

The Why tool generates proof obligations written in a first-order multi-sorted theory, using the Why primitive operations on basic types such as integers or booleans and also model-specific symbols for constants, functions and predicates. In order to prove properties involving these symbols, we provide an axiomatic first-order theory, used to discharge the proof obligations with an automatic prover such as Simplify. We developed a Coq realization of that theory. This model uses higher-order constructions for representing the memories operations.

If we assume that deduction steps performed by Simplify are correct, then they could be translated into Coq, leading to a complete proof in Coq of the original proof obligations.

By designing a suitable modeling of Java memory states, together with a static computation of effects and a suitable background first-order theory, we obtained a powerful method for proving functional properties of Java programs. Combined with an automatic theorem prover, we are able to establish automatically more properties than static checkers like ESC/Java or ChAsE, with a method whose soundness only rely on the soundness of the translation provided in Section 4.

We believe we made a significant step in filling the gap between static checking techniques, fully automatic but unsound, and true formal verification which requires user interaction: our approach is a compromise between safety of the global approach and push-button technology.

We took advantage of the modular architecture of Why which does all the work of generation of proof obligations for different provers. We believe that this modular architecture is a good approach, that can be easily reused for different input languages than Java. For example we have been able to build, in a quite short time, a similar modeling for C programs [12], with full support for pointer arithmetic.

7.2 Related Work

In [17], F. Mehta and T. Nipkow used the Isabelle proof assistant in order to prove an imperative program involving pointers using a model similar to ours, but without arrays nor memory allocation.

Several tools exist which manipulate Java programs annotated with JML specifications [6]. Their objectives can be different, they may aim at producing code with dynamic testing, or generating programs for unit testing of classes, or proving properties of programs. We already mentioned ESC/Java which is fully automatic but does not guaranty correctness. The memory modeling of ESC/Java seems similar to ours. LOOP [22, 23], Jive [18], JACK [5] or our tool Krakatoa are intended to generate for any JML specification of the program, sufficient verification conditions for these properties to hold. These tools are based on different techniques: both Jive and LOOP use a global memory modeling ; Jive is based on a weakest precondition generator ; in LOOP, the semantics of JML-annotated Java programs is translated into functional PVS expressions which represent the denotational semantics of the program, and properties of these programs can be established using specialized PVS tactics. The JACK environment [5], initially developed by the Gemplus company and now by INRIA, uses a memory model similar to ours, and was initially designed for generating proof obligations for the B system [1] but now also has an output for Simplify.

7.3 Future Work

Automatic provers are useful for early detection of errors in code or specification. We plan to be able to analyze counter-examples in order to suggest proof

annotations. A partial analysis of correctness of loops could also help in finding appropriate loop invariants.

One very interesting future work is to be able to build, with the underlying automatic prover, a proof trace which could be double-checked by a proof assistant: in this way, only proof obligations that cannot be solved automatically would need to be proved manually. To obtain such a trace, the use of the HARVEY [20] and CVC-lite[3] tools is currently under investigation.

There are still important JAVA and JML features not yet supported by the KRAKATOA tool. Handling the class invariants may become heavy when they are many objects involved, and their combination with inheritance causes important theoretical issues [2].

Acknowledgements. We thank Gary T. Leavens and David Cok for their useful comments on a preliminary version of this paper.

References

[1] Jean-Raymond Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[3] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, Boston, MA, USA, July 2004. Springer-Verlag.

[4] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] Lilian Burdy. JACK: Java Applet Correctness Kit. Gemplus Developer Conference, 2002.

[6] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Technical Report NIII-R0309, Dept. of Computer Science, University of Nijmegen, 2003.

[7] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[8] Néstor Cataño, M. Gawlowski, Marieke Huisman, Bart Jacobs, Claude Marché, Christine Paulin, Erik Poll, Nicole Rauch, and Xavier Urbain. Logical techniques for applet verification. Deliverable 5.2, IST VerifiCard project, 2003. `http://www.cs.kun.nl/VerifiCard/files/deliverables/deliverable_5_2.pdf`.

[9] N. Cataño and M. Huisman. Chase: A static checker for JML's assignable clause. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Proc. VMCAI*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40, New York, USA, January 2003. Springer-Verlag.

[10] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[11] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[12] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, November 2004. Springer-Verlag.

[13] Bart Jacobs, Claude Marché, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, Stirling, UK, July 2004. Springer-Verlag.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

[15] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, and Clyde Ruby. *JML Reference Manual*, April 2003. draft.

[16] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[17] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In Franz Baader, editor, *19th Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[18] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system. `http://www.informatik.fernuni-hagen.de/pi5/publications.html`, 2000.

[19] Greg Nelson. Techniques for program verification. Research Report CSL-81-10, Xerox Palo Alto Research Center, 1981. `http://research.compaq.com/SRC/esc/Simplify.html`.

[20] Silvio Ranise and David Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *Proc. SEFM'03*, Canberra, Australia, September 2003. IEEE Computer Society Press. `http://www.loria.fr/equipes/cassis/softwares/haRVey/`.

[21] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

[22] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2000.

[23] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Proc. TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer-Verlag, 2001. `http://www.cs.kun.nl/~bart/LOOP`.

Real Number Calculations and Theorem Proving

César Muñoz[1] and David Lester[2]

[1] National Institute of Aerospace, 144 Research Drive, Hampton VA 23666, USA
munoz@nianet.org
[2] University of Manchester, Oxford Road, Manchester M13 9PL, UK
dlester@cs.man.ac.uk

Abstract. Wouldn't it be nice to be able to *conveniently* use ordinary real number expressions within proof assistants? In this paper we outline how this can be done within a theorem proving framework. First, we formally establish upper and lower bounds for trigonometric and transcendental functions. Then, based on these bounds, we develop a rational interval arithmetic where real number calculations can be performed in an algebraic setting. This pragmatic approach has been implemented as a strategy in PVS. The strategy provides a safe way to perform explicit calculations over real numbers in formal proofs.

1 Introduction

In the verification of an engineering application it is often necessary to perform explicit calculations on non-algebraic functions. Despite all the developments on real analysis in theorem provers [12, 7, 8, 15, 9], formal justification of these calculations is not routine.

Take, for example, the formula

$$\frac{3\pi}{180} \leq \frac{g}{v}\tan(\frac{35\pi}{180}), \tag{1}$$

where g is the gravitational force and $v = 250$ kt is the ground speed of an aircraft. This formula appears in the verification of the NASA's Airborne Information for Lateral Spacing (AILS) algorithm presented in [18]. It states that the maximum turn rate of an aircraft flying at ground speed v with a bank angle of 35^o is 3^o per second. The original proof is about a dozen lines and requires the use of several trigonometric properties.

In some cases the formal checking of a numerical inequality is so cumbersome that the effort seems futile; it is then tempting to perform the calculation out of the system, and introduce the result as an axiom[1]. However, the chances are that the external calculation will be performed using floating-point arithmetic. Without formal checking of the result, we will never be sure of the correctness of the calculation.

[1] As a matter of fact, the initial verification of NASA's AILS algorithm contained several of such axioms.

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 195–210, 2005.

In this paper we present a method to automatically prove numerical inequalities, such as Formula (1), within a proof assistant. The point of departure is a collection of lower and upper bounds for rational and non-rational operations. Based on provable properties of these bounds, we develop a rational interval arithmetic which is amenable to automation. The series approximations and interval arithmetic used here are well-known. However, to our knowledge, this is the most complete formalization of exact real arithmetic and interval arithmetic within a theorem prover.

The rest of this paper is organized as follows. Section 2 defines bounds for square root and transcendental functions. Section 3 presents a rational interval arithmetic based on these bounds. Section 4 describes a method to prove numerical inequalities. The implementation of this method in PVS is described in Section 5. Last section summarizes our work and compares it to related work. For readability, we will use standard mathematical notation along this paper. However, we remark that the mathematical development presented in this paper has been written and fully verified in PVS. Furthermore, all the development is freely available on the Internet. The results on upper and lower bounds have been integrated to the NASA Langley PVS Libraries at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`. The rational interval arithmetic and the PVS strategy for numerical inequalities are available from `http://research.nianet.org/~munoz/Interval`.

2 Bounds for Square Root and Transcendental Functions

A PVS basic theory of bounds for square root and trigonometric functions was originally proposed for the verification of an algorithm for aircraft conflict detection [18]. It has been completed and extended with bounds for natural logarithm, exponential, and arctangent. The basic idea is to provide for each real function $f : \mathbb{R} \mapsto \mathbb{R}$, functions $\underline{f} : (\mathbb{R}, \mathbb{N}) \mapsto \mathbb{R}$ and $\overline{f} : (\mathbb{R}, \mathbb{N}) \mapsto \mathbb{R}$ closed under $\mathbb{Q}$, such that for all x, n

$$\underline{f}(x,n) \;\leq\; f(x) \;\leq\; \overline{f}(x,n), \tag{2}$$

$$\underline{f}(x,n) \;\leq\; \underline{f}(x,n+1), \tag{3}$$

$$\overline{f}(x,n+1) \;\leq\; \overline{f}(x,n), \tag{4}$$

$$\lim_{n\to\infty} \underline{f}(x,n) \;=\; f(x) \;=\; \lim_{n\to\infty} \overline{f}(x,n). \tag{5}$$

Formula (2) states that $\underline{f}$ and $\overline{f}$ are, respectively, lower and upper bounds of f, and formulas (3), (4), and (5) state that these bounds can be improved, as much as needed, by increasing the approximation parameter n.

For transcendental functions, we use Taylor's approximation series. Because the convergence is usually best for a small range of values, we have used Taylor's Theorem only on a small range, and then exploited the technique of *range reduction*. All the stated propositions in this section have been formally verified in PVS.

2.1 Square Root

For square root, we use a simple approximation by Newton's method. For $x \geq 0$,

$$\begin{aligned} \overline{\mathrm{sqrt}}(x,0) &= x+1, \\ \overline{\mathrm{sqrt}}(x,n+1) &= \frac{1}{2}(y+\frac{x}{y}), \quad \text{where } y = \overline{\mathrm{sqrt}}(x,n), \\ \underline{\mathrm{sqrt}}(x,n) &= \frac{x}{\overline{\mathrm{sqrt}}(x,n)}. \end{aligned}$$

Proposition 1. $\forall x,n:\ x \geq 0 \ \Rightarrow\ 0 \ \leq\ \underline{\mathrm{sqrt}}(x,n) \ \leq\ \sqrt{x} \ <\ \overline{\mathrm{sqrt}}(x,n)$.

The first inequality is strict when $x > 0$.

2.2 Trigonometric Functions

We use the partial approximation by series.

$$\begin{aligned} \underline{\sin}(x,n) &= \sum_{i=1}^{m}(-1)^{i-1}\frac{x^{2i-1}}{(2i-1)!} \\ \overline{\sin}(x,n) &= \sum_{i=1}^{m+1}(-1)^{i-1}\frac{x^{2i-1}}{(2i-1)!}, \\ \underline{\cos}(x,n) &= 1+\sum_{i=1}^{m+1}(-1)^{i}\frac{x^{2i}}{(2i)!}, \\ \overline{\cos}(x,n) &= 1+\sum_{i=1}^{m}(-1)^{i}\frac{x^{2i}}{(2i)!}, \end{aligned}$$

where $m = 2n$ if $x < 0$; otherwise, $m = 2n+1$.

Proposition 2. $\forall x,n:\ \underline{f}(x,n) \ \leq\ f(x) \ \leq\ \overline{f}(x,n)$, *for* $f \in \{\sin,\cos\}$.

2.3 Arctangent and π

We first use the alternating partial approximation by series for $0 \leq x \leq 1$.

$$\begin{aligned} \underline{\mathrm{atan}}(x,n) &= \sum_{i=1}^{2n+1} x^{2i+1}\frac{(-1)^i}{2i+1}, \quad \text{if } 0 < x \leq 1, \\ \overline{\mathrm{atan}}(x,n) &= \sum_{i=1}^{2n} x^{2i+1}\frac{(-1)^i}{2i+1}, \quad \text{if } 0 < x \leq 1. \end{aligned}$$

We note that for $x = 1$ (which we might naïvely wish to use to define $\pi/4$ and hence π) the series: $1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots$ *does* converge, but very slowly. Instead

we use the equality $\frac{\pi}{4} = 4\ \mathrm{atan}(1/5) - \mathrm{atan}(1/239)$, which then has much better convergence properties. Using this identity we can define bounds on π:

$$\underline{\pi}(n) = 16\ \underline{\mathrm{atan}}(1,n) - 4\ \overline{\mathrm{atan}}(1,n),$$
$$\overline{\pi}(n) = 16\ \overline{\mathrm{atan}}(1,n) - 4\ \underline{\mathrm{atan}}(1,n).$$

Proposition 3. $\forall n :\ \underline{\pi}(n) \ \leq\ \pi\ \leq\ \overline{\pi}(n).$

Now, we extend the range of the arctangent function to the whole set of real numbers:

$$\underline{\mathrm{atan}}(0,n) = \overline{\mathrm{atan}}(0,n)\ =\ 0,$$
$$\underline{\mathrm{atan}}(x,n) = \frac{\underline{\pi}(n)}{2} - \overline{\mathrm{atan}}(\frac{1}{x},n), \quad \text{if } 1 < x,$$
$$\underline{\mathrm{atan}}(x,n) = -\overline{\mathrm{atan}}(-x,n), \quad \text{if } x < 0,$$
$$\overline{\mathrm{atan}}(x,n) = \frac{\overline{\pi}(n)}{2} - \underline{\mathrm{atan}}(\frac{1}{x},n), \quad \text{if } 1 < x,$$
$$\overline{\mathrm{atan}}(x,n) = -\underline{\mathrm{atan}}(-x,n), \quad \text{if } x < 0.$$

Proposition 4. $\forall x, n :\ \underline{\mathrm{atan}}(x,n)\ \leq\ \mathrm{atan}(x)\ \leq\ \overline{\mathrm{atan}}(x,n).$

These are strict inequalities except when $x = 0$.

2.4 Exponential

The fundamental series we use for the exponential function is

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

We could directly find bounds for negative x from this series as, in this case, the series is alternating. However, we will subsequently find that it is convenient to show that our bounds for the exponential function are strictly positive, and with the above bounds this is not true. It *is* true for $-1 \leq x \leq 0$. This is proven by using Taylor's Theorem for the exponential function on the range $-1 \leq x < 0$. Therefore, we define

$$\underline{\exp}(x,n) = \sum_{i=1}^{2n+1} \frac{x^i}{i!}, \quad \text{if } -1 \leq x < 0,$$
$$\overline{\exp}(x,n) = \sum_{i=1}^{2n} \frac{x^i}{i!}, \quad \text{if } -1 \leq x < 0.$$

Using properties of the exponential function, we obtain bounds for the whole set of real numbers:

$$\begin{aligned}
\underline{\exp}(0,n) &= \overline{\exp}(0,n) \;=\; 1,\\
\underline{\exp}(x,n) &= \underline{\exp}(x/m,n+1)^m, \quad \text{if } x \le -1,\\
\underline{\exp}(x,n) &= \frac{1}{\overline{\exp}(-x,n)}, \quad \text{if } x > 0,\\
\overline{\exp}(x,n) &= \overline{\exp}(x/m,n+1)^m, \quad \text{if } x \le -1.\\
\overline{\exp}(x,n) &= \frac{1}{\underline{\exp}(-x,n)}, \quad \text{if } x > 0.
\end{aligned}$$

Notice that unless we can ensure that all of the bounding functions are strictly positive we will run into type-checking problems using the bound definitions for $x > 0$, e.g., $1/\overline{\exp}(-x,n)$ is only defined provided $\overline{\exp}(-x,n) \neq 0$.

Proposition 5. $\forall x, n:\ 0 \;<\; \underline{\exp}(x,n) \;\le\; \exp(x) \;\le\; \overline{\exp}(x,n)$.

These are strict inequalities except when $x = 0$.

2.5 Natural Logarithm

For $-1 < x \le 1$, we use the alternating series for natural logarithm:

$$\ln(x+1) = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{x^i}{i}.$$

Therefore, we define

$$\begin{aligned}
\underline{\ln}(x,n) &= \sum_{i=1}^{2n} (-1)^{i+1} \frac{(x-1)^i}{i}, \quad \text{if } 1 < x \le 2,\\
\overline{\ln}(x,n) &= \sum_{i=1}^{2n+1} (-1)^{i+1} \frac{(x-1)^i}{i}, \quad \text{if } 1 < x \le 2.
\end{aligned}$$

Using properties of the natural logarithm function, we obtain

$$\begin{aligned}
\underline{\ln}(1,n) &= \overline{\ln}(1,n) \;=\; 1,\\
\underline{\ln}(x,n) &= -\underline{\ln}(\frac{1}{x},n), \quad \text{if } 0 < x < 1,\\
\overline{\ln}(x,n) &= -\overline{\ln}(\frac{1}{x},n), \quad \text{if } 0 < x < 1.
\end{aligned}$$

Finally, we extend the range to the whole set of positive reals. If $x > 2$, we find a natural number m and real number y such that $x = 2^m y$ and $1 < y \le 2$, by using an algorithm similar to Euclidean division. Then, we observe that

$$\ln(x) = \ln(2^m y) = m \ln(2) + \ln(y).$$

Hence,

$$\begin{aligned}
\underline{\ln}(x,n) &= m\ \underline{\ln}(2,n) + \underline{\ln}(y,n), \quad \text{if } x > 2,\\
\overline{\ln}(x,n) &= m\ \overline{\ln}(2,n) + \overline{\ln}(y,n), \quad \text{if } x > 2.
\end{aligned}$$

Proposition 6. $\forall x, n: \ 0 < x \ \Rightarrow \ \underline{\ln}(x,n) \ \leq \ \ln(x) \ \leq \ \overline{\ln}(x,n)$.

These are strict inequalities except when $x = 1$.

3 Rational Interval Arithmetic

A *(closed) interval* $[a, b]$ is the set of real numbers between a and b, i.e.,

$$[a, b] = \{x \mid a \leq x \leq b\}.$$

The bounds a and b are called the *lower bound* and *upper bound* of $[a, b]$, respectively. Note that if $a > b$, the interval is empty. The notation $[a]$ abbreviates the point-wise interval $[a, a]$.

Since interval computations are mostly performed on the bounds of the intervals, the set where these bounds are defined is critical. This set is called the *base number type*. Systems for interval analysis and exact arithmetic usually consider the base number type to be machine floating-point numbers. In general, interval computations assume a correct implementation of the IEEE 754 standard [13]. For this work, we take a different approach where the base number type is the set of rational numbers. Trigonometric and transcendental functions for interval arithmetic are defined using the parameterizable bounding functions presented in Section 2.

In the following, we use the first letters of the alphabet $a, b, \ldots$ to denote rational numbers, and the last letters of the alphabet $\ldots x, y, z$ to denote arbitrary real variables. We use **boldface** for interval variables. If $\mathbf{x}$ is an interval variable, $\underline{\mathbf{x}}$ denotes its lower bound and $\overline{\mathbf{x}}$ denotes its upper bound. The four basic interval operations are defined as follows [14]:

$$\begin{aligned}
\mathbf{x} + \mathbf{y} &= [\underline{\mathbf{x}} + \underline{\mathbf{y}}, \overline{\mathbf{x}} + \overline{\mathbf{y}}], \\
\mathbf{x} - \mathbf{y} &= [\underline{\mathbf{x}} - \overline{\mathbf{y}}, \overline{\mathbf{x}} - \underline{\mathbf{y}}], \\
\mathbf{x} \times \mathbf{y} &= [\min\{\underline{\mathbf{x}}\underline{\mathbf{y}}, \underline{\mathbf{x}}\overline{\mathbf{y}}, \overline{\mathbf{x}}\underline{\mathbf{y}}, \overline{\mathbf{x}}\overline{\mathbf{y}}\}, \max\{\underline{\mathbf{x}}\underline{\mathbf{y}}, \underline{\mathbf{x}}\overline{\mathbf{y}}, \overline{\mathbf{x}}\underline{\mathbf{y}}, \overline{\mathbf{x}}\overline{\mathbf{y}}\}], \\
\mathbf{x} \div \mathbf{y} &= \mathbf{x} \times [\frac{1}{\overline{\mathbf{y}}}, \frac{1}{\underline{\mathbf{y}}}], \quad \text{if } \underline{\mathbf{y}}\overline{\mathbf{y}} > 0.
\end{aligned}$$

We also define the unary negation, absolute value, and power operators for intervals:

$$\begin{aligned}
-\mathbf{x} &= [-\overline{\mathbf{x}}, -\underline{\mathbf{x}}], \\
|\mathbf{x}| &= [\min\{|\underline{\mathbf{x}}|, |\overline{\mathbf{x}}|\}, \max\{|\underline{\mathbf{x}}|, |\overline{\mathbf{x}}|\}], \quad \text{if } \underline{\mathbf{x}}\overline{\mathbf{x}} \geq 0. \\
|\mathbf{x}| &= [0, \max\{|\underline{\mathbf{x}}|, |\overline{\mathbf{x}}|\}], \quad \text{if } \underline{\mathbf{x}}\overline{\mathbf{x}} < 0. \\
\mathbf{x}^n &= \begin{cases} [1] & \text{if } n = 0, \\ [\underline{\mathbf{x}}^n, \overline{\mathbf{x}}^n] & \text{if } \underline{\mathbf{x}} \geq 0 \ \text{ or odd?}(n), \\ [\overline{\mathbf{x}}^n, \underline{\mathbf{x}}^n] & \text{if } \overline{\mathbf{x}} \leq 0 \ \text{ and even?}(n), \\ [0, \max\{\underline{\mathbf{x}}^n, \overline{\mathbf{x}}^n\}] & \text{otherwise.} \end{cases}
\end{aligned}$$

Interval operations are defined such that they include the result of their corresponding real operations. This property is called the *inclusion property* and is formally expressed as follows:

Proposition 7. *If $x \in \mathbf{x}$ and $y \in \mathbf{y}$ then $x \otimes y \in \mathbf{x} \otimes \mathbf{y}$, where $\otimes \in \{+, -, \times, \div\}$. Moreover, $-x \in -\mathbf{x}$, $|x| \in |\mathbf{x}|$, and $x^n \in \mathbf{x}^n$, for $n \geq 0$. It is assumed that $\mathbf{y}$ does not contain 0 in the case of interval division.*

The inclusion property is fundamental to interval arithmetic. It guarantees that the evaluations of an expression using interval arithmetic is a correct approximation of the exact real value. Any operation in interval arithmetic must satisfy the inclusion property with respect to the corresponding real operation.

3.1 Interval Comparisons

There are several possible ways to compare intervals [27]. In this work, we use interval-rational comparisons and interval inclusions.

$$\begin{aligned} \mathbf{x} \lhd a \quad & \text{if } \overline{\mathbf{x}} \lhd a, \text{ for } \lhd \in \{<, \leq\}, \\ \mathbf{x} \rhd a \quad & \text{if } \underline{\mathbf{x}} \rhd a, \text{ for } \rhd \in \{>, \geq\}, \\ \mathbf{x} \subseteq \mathbf{y} \quad & \text{if } \underline{\mathbf{y}} \leq \underline{\mathbf{x}} \text{ and } \overline{\mathbf{x}} \leq \overline{\mathbf{y}}. \end{aligned}$$

Proposition 8. *Assume that $x \in \mathbf{x}$,*

1. *if $\mathbf{x} \bowtie a$ then $x \bowtie a$, for $\bowtie \in \{<, \leq, >, \geq\}$, and*
2. *if $\mathbf{x} \subseteq \mathbf{y}$ then $x \in \mathbf{y}$.*

We use $\not\bowtie$ to denote $\geq$, $>$, $\leq$, or $<$, when $\bowtie$ is, respectively, $<$, $\leq$, $>$, or $\geq$.

Proposition 9. *If $\mathbf{x} \bowtie a$ and $\mathbf{x} \not\bowtie a$, then $\mathbf{x}$ is empty.*

Notice that $\neg(\mathbf{x} \bowtie a)$ does not imply $\mathbf{x} \not\bowtie a$. For instance, $[-1, 1]$ is neither greater nor less than 0.

3.2 Square Root, Arctangent, Exponential, and Natural Logarithm

Interval functions for square root, arctangent, exponential, and natural logarithm are defined for an approximation parameter $n \geq 0$:

$$\begin{aligned} [\sqrt{\mathbf{x}}]_n &= [\underline{\mathrm{sqrt}}(\underline{\mathbf{x}}, n), \overline{\mathrm{sqrt}}(\overline{\mathbf{x}}, n)], \quad \text{if } 0 \leq \underline{\mathbf{x}} \leq \overline{\mathbf{x}}, \\ [\mathrm{atan}(\mathbf{x})]_n &= [\underline{\mathrm{atan}}(\underline{\mathbf{x}}, n), \overline{\mathrm{atan}}(\overline{\mathbf{x}}, n)], \\ [\exp(\mathbf{x})]_n &= [\underline{\exp}(\underline{\mathbf{x}}, n), \overline{\exp}(\overline{\mathbf{x}}, n)], \\ [\ln(\mathbf{x})]_n &= [\underline{\ln}(\underline{\mathbf{x}}, n), \overline{\ln}(\overline{\mathbf{x}}, n)], \quad \text{if } 0 < \underline{\mathbf{x}} \leq \overline{\mathbf{x}}. \end{aligned}$$

The above functions satisfy the following inclusion property.

Proposition 10. *If $x \in \mathbf{x}$ then $f(x) \in [f(\mathbf{x})]_n$, where $f \in \{\sqrt{}, \mathrm{atan}, \exp, \ln\}$. It is assumed that $\mathbf{x}$ is non-negative in the case of square root, and $\mathbf{x}$ is positive in the case of natural logarithm.*

Proof. This is a consequence of Propositions 1, 4, 5, and 6 in Section 2, and the fact that these functions are increasing. □

3.3 Trigonometric Functions

Parametric trigonometric functions for intervals are defined as follows:

$$[\sin(\mathbf{x})]_n = \begin{cases} [\underline{\sin}(\underline{\mathbf{x}}, n), \overline{\sin}(\overline{\mathbf{x}}, n)] & \text{if } \mathbf{x} \subseteq [-\frac{\underline{\pi}(n)}{2}, \frac{\underline{\pi}(n)}{2}], \\ [\underline{\sin}(\overline{\mathbf{x}}, n), \overline{\sin}(\underline{\mathbf{x}}, n)] & \text{if } \mathbf{x} \subseteq [\frac{\overline{\pi}(n)}{2}, \underline{\pi}(n)], \\ [\underline{\sin}(\underline{\mathbf{x}}, n), \overline{\sin}(\overline{\mathbf{x}}, n)] & \text{if } \mathbf{x} \subseteq [0, \frac{\underline{\pi}(n)}{2}], \\ -[\sin(-\mathbf{x})]_n & \text{if } \mathbf{x} \subseteq [-\underline{\pi}(n), 0], \\ [-1, 1] & \text{otherwise}, \end{cases} \tag{6}$$

$$[\cos(\mathbf{x})]_n = \begin{cases} [\underline{\cos}(\overline{\mathbf{x}}, n), \overline{\cos}(\underline{\mathbf{x}}, n)] & \text{if } \mathbf{x} \subseteq [0, \underline{\pi}(n)], \\ [\cos(-\mathbf{x})]_n & \text{if } \mathbf{x} \subseteq [-\underline{\pi}(n), 0], \\ [\min\{\underline{\cos}(\underline{\mathbf{x}}, n), \underline{\cos}(\overline{\mathbf{x}}, n)\}, 1] & \text{if } \mathbf{x} \subseteq [-\frac{\underline{\pi}(n)}{2}, \frac{\underline{\pi}(n)}{2}], \\ [-1, 1] & \text{otherwise}, \end{cases} \tag{7}$$

$$[\tan(\mathbf{x})]_n = [\frac{\underline{\sin}}{\overline{\cos}}(\underline{\mathbf{x}}, n+3), \frac{\overline{\sin}}{\underline{\cos}}(\overline{\mathbf{x}}, n+3)], \quad \text{if } \mathbf{x} \subseteq [-\frac{\underline{\pi}(n)}{2}, \frac{\underline{\pi}(n)}{2}]. \tag{8}$$

The $n + 3$ in Formula (8) is necessary to guarantee that the lower and upper bounds of cosine are always positive. Therefore, the tangent function is well-defined in the interval $[-\frac{\underline{\pi}(n)}{2}, \frac{\underline{\pi}(n)}{2}]$.

The above functions satisfy the following inclusion property.

Proposition 11. *If $x \in \mathbf{x}$ then $f(x) \in [f(\mathbf{x})]_n$, where $f \in \{\sin, \cos\}$. Moreover, if $\mathbf{x} \subseteq [-\frac{\underline{\pi}(n)}{2}, \frac{\underline{\pi}(n)}{2}]$, $\tan(x) \in [\tan(\mathbf{x})]_n$.*

Proof. This is a consequence of Proposition 2 in Section 2, and a case analysis on the quadrant where the functions are increasing or decreasing. □

The next section proposes a method to prove numerical inequalities based on the rational interval arithmetic described here.

4 Proving Numerical Inequalities

Arithmetic expressions are defined by the following grammar, where $\mathcal{V}$ is a denumerable set of real variables:

$$\begin{array}{lcl} e & ::= & a \mid x \mid e + e \mid e - e \mid -e \mid e \times e \mid e \div e \mid |e| \mid e^i \mid \sqrt{e} \mid \\ & & \pi \mid \sin(e) \mid \cos(e) \mid \tan(e) \mid \exp(e) \mid \ln(e) \mid \operatorname{atan}(e) \\ a & \in & \mathbb{Q} \\ i & \in & \mathbb{N} \\ x & \in & \mathcal{V} \end{array}$$

As usual, parenthesis are used to group subexpressions as needed.

A *context* Γ is a set of couples $(x, \mathbf{x})$ where x is a real variable and $\mathbf{x}$ is a non-empty interval. The intended semantics of contexts is given by the following deduction rule in the sequent calculus style:

$$\frac{(x, \mathbf{x}) \in \Gamma}{\Gamma \vdash x \in \mathbf{x}}.$$

Given a real expression e, a context Γ that includes all variables in e, and an approximation parameter n, the interval expression $[e]_n^\Gamma$ is recursively defined as follows:

$$\begin{aligned}
[a]_n^\Gamma &= [a], \\
[x]_n^\Gamma &= \mathbf{x}, \quad \text{where } (x, \mathbf{x}) \in \Gamma, \\
[e_1 \otimes e_2]_n^\Gamma &= [e_1]_n^\Gamma \otimes [e_2]_n^\Gamma, \quad \text{where } \otimes \in \{+, -, \times, \div\}, \\
[e^i]_n^\Gamma &= ([e]_n^\Gamma)^i, \\
[-e]_n^\Gamma &= -[e]_n^\Gamma, \\
[|e|]_n^\Gamma &= |[e]_n^\Gamma|, \\
[\pi]_n^\Gamma &= [\underline{\pi}(n), \overline{\pi}(n)], \\
[f(x)]_n^\Gamma &= [f([x]_n^\Gamma)]_n, \quad \text{where } f \in \{\sin, \cos, \tan, \exp, \ln, \text{atan}\}.
\end{aligned}$$

Proposition 12. *Let e be an arithmetic expression, Γ be a context that includes all variables in e, n an approximation parameter, and $\mathbf{e} = [e]_n^\Gamma$. Assume that e and $\mathbf{e}$ are well-defined, i.e., side conditions are satisfied for division, square root, logarithm, and tangent for real and interval values. Therefore,*

$$\Gamma \ \vdash \ e \in \mathbf{e}. \tag{9}$$

Proof. By structural induction on e and propositions 3, 7, 10, and 11.

4.1 A General Method

We propose the following general method to prove the sequent

$$\Gamma \ \vdash \ e_1 \bowtie e_2,$$

where e_1 and e_2 are well-defined arithmetic expressions, and $\bowtie \in \{<, \leq, >, \geq\}$:

1. Select an approximation parameter n.
2. Define $e = e_1 - e_2$.
3. Define $\mathbf{e} = [e]_n^\Gamma$ and show that it is well-defined.
4. By Proposition 12,
$$\Gamma \vdash e \in \mathbf{e}.$$
5. Evaluate $\mathbf{e} \bowtie 0$. If it evaluates to true, it means that the following sequent holds
$$\Gamma \vdash \mathbf{e} \bowtie 0.$$
In that case go to step 6. In the other case, evaluate $\mathbf{e} \not\bowtie 0$. If this evaluates to true then fail. By Proposition 9, the sequent $\Gamma \ \vdash \ \mathbf{e} \bowtie 0$ cannot hold. If $\mathbf{e} \not\bowtie 0$ evaluates to false, increase the approximation parameter and return to step 3.
6. Proposition 8 yields
$$\Gamma \vdash e \bowtie 0.$$

7. By definition,

$$\Gamma \vdash e_1 - e_2 \bowtie 0.$$

8. Therefore,

$$\Gamma \vdash e_1 \bowtie e_2.$$

The method above is a *sound*, i.e., all the steps can be effectively computed and each one is formally justified. In particular, well-definedness of $\mathbf{e}$ and the inequalities $\mathbf{e} \bowtie 0$ and $\mathbf{e} \not\bowtie 0$ can be mechanically checked as they only involve rational arithmetic. However, the method is not *complete* as the it does not necessarily terminate. Even if e only involves the four basic operations, it may be that both $\mathbf{e} \bowtie 0$ and $\mathbf{e} \not\bowtie 0$ evaluate to false.

The absence of a completeness result is a fundamental limitation on any general computable arithmetic. At a practical level, the problem arises because all we have available are a sequence of approximations to the real numbers x and y; provided x and y differ, with luck we will eventually have a pair of approximations whose intervals do not overlap, and hence we can return a result for $x \bowtie y$. However, if x and y are the same real number (note we might not necessarily get the same sequence of approximations for both x and y), we can never be sure whether further evaluation might result in us being able to distinguish the numbers. Theoreticians might prefer the statement that to be computable, a function must at least be (computably) continuous, and that any attempt to define non-constant continuous functions from the (computable) reals to the booleans is futile [21, 22, 24, 25].

4.2 Sub-distributive Arithmetic

Interval arithmetic is sub-distributive, i.e., $\mathbf{x} \times (\mathbf{y} + \mathbf{z}) \subseteq \mathbf{x} \times \mathbf{y} + \mathbf{x} \times \mathbf{z}$. In the general case the inclusion is strict. This effect is also called *decorrelation* and it is due to the fact that interval identity is lost in interval arithmetic. This may have surprising effects, for instance $\mathbf{x} - \mathbf{x}$ is $[0]$ only if $\mathbf{x}$ is point-wise, e.g., $[0,1] - [0,1] = [-1,1]$. Moreover, as we have seen in Section 3.1, both $\mathbf{x} \geq a$ and $\mathbf{x} < a$ may be false.

For the method presented in the previous section, it means that the arrangement of the expression e matters. For instance, assume that we want to prove $\Gamma \vdash 2 \times x \geq x$ assuming that $x \in [0,1]$ is in Γ. This is pretty obvious in arithmetic as x is a non-negative real. Using our method, we consider the arithmetic expression $e = 2 \times x - x$ and construct the interval expression $\mathbf{e} = 2 \times \mathbf{x} - \mathbf{x}$, where $\mathbf{x} = [0,1]$. For any approximation parameter, $\mathbf{e}$ evaluates to $[-1,2]$ which is neither greater nor less than 0. Therefore, the method will not terminate. This effect may happen even if e is ground.

On the other hand, if instead of the arithmetic expression $2 \times x - x$, we consider the equivalent arithmetic expression $(2-1) \times x$, the corresponding interval property $([2] - [1]) \times \mathbf{x}$ evaluates to $[0,1]$ which is non-negative.

4.3 The Bounds of π

Note that sin and cos are defined for the whole real line. However, for angles α such that $|\alpha| \geq \pi$ both functions will return the interval $[-1, 1]$, a valid approximation but not a very good one.

Even for angles less than π, the bounds computed by formulas (6) and (7) may not be very accurate. For example, consider the arithmetic expression $e = \sin(\frac{\pi}{2})$ and, for an approximation parameter n, the corresponding interval expression $\mathbf{e} = [\sin([\frac{\underline{\pi}(n)}{2}, \frac{\overline{\pi}(n)}{2}])]_n$. From the definition of $[\sin(\mathbf{x})]_n$, we get $\mathbf{e} = [-1, 1]$, as the interval $[\frac{\underline{\pi}(n)}{2}, \frac{\overline{\pi}(n)}{2}]$ falls in the default case. Therefore, the fact that $\sin(\frac{\pi}{2}) > 0$, cannot be proven using our method.

4.4 Symbolic Evaluation

Our method relies on explicit calculations to check for well-definedness of interval expressions in step 3 and to verify interval inequalities in step 5. In theorem provers, explicit calculations usually means symbolic evaluations, which are extremely inefficient for the interval functions that we want to calculate.

Section 5 describes how all these issues are handled in the PVS Interval package that we have developed.

5 Automation in PVS

The interval arithmetic presented in this paper has been formalized and verified in PVS [20]. It is available as a PVS package called Interval. The strategy `numerical`, which is part of the Interval package, implements the method described in Section 4.1. This strategy automatically discharges sequent of the forms $\Gamma \vdash e_1 \bowtie e_2$, for $\bowtie \in \{<, \leq, >, \geq\}$, and $\Gamma \vdash e \in [a, b]$.

Actual definitions in PVS have been slightly modified for efficiency reasons. For instance, multiplication is defined using a case analysis on the sign of the operands. Additionally, all interval operations are completed by returning an empty interval if side conditions are not satisfied. This technique avoids the generation of type correctness conditions in some instances.

The strategy `numerical` is aimed to practicality rather than accuracy. For example, it might not be able to prove that $\sqrt{4} \in [2]$, but it can prove that $\sqrt{4} \in [1.5, 2.5]$, or, even better, that $\sqrt{4} \in [1.99, 2.01]$. With this in mind, we designed a strategy that:

- Always terminates (in a reasonable period of time).
- Works over the PVS built-in type `real` (in contrast to a strategy for a new data type of arithmetic expressions).
- Is configurable for better accuracy (at the expense of performance).

5.1 Termination

Termination is trivially achieved as the strategy does not iterate for different approximations, i.e., step 5 either goes to step 6 or fails. In other words, if

`numerical` does not succeed, it does nothing. By default, `numerical` picks a default approximation value of 3 which gives an accuracy of about 2 decimals for trigonometric functions. However, the user can increase this parameter or set a different approximation to each function according to its accuracy needs and availability of computational power. Currently, there is no direct relation between the approximation parameter and the degree of the accuracy, as all the bounding functions have different convergence rates. On-going work aims to provide an absolute error of at most 10^{-n} for the approximation parameter n. This will give additional control to the user on the accuracy of the result. However, this technique will not guarantee the precision of the final result as computation errors are accumulative. Automatic iteration of the method for continuously increasing approximation parameters is not supported as the strategies have not being designed to reuse past computations. Without a reusing mechanism it will be prohibitively expensive to automatically iterate `numerical` to achieve a small approximation on a complex arithmetic expression.

5.2 Data Type for Real Numbers

The strategies in the Interval package work over the PVS built-in real numbers. The major advantage of this approach is that the functionality of the strategies can be extended to handle user defined real operators and functions without modifying the strategy code. Indeed, optional parameters to the strategy `numerical` allow for the specification of arbitrary real functions. If the interval interpretations are not provided, the strategy tries to build them from the syntactic definition of the functions. The trade-off for the use of the PVS type `real`, in favor of a defined data type for arithmetic expressions, is that the function $[e]_n^\Gamma$ and Proposition 12 are at the meta-level, i.e., they are not written in PVS. It also means that the soundness of our method cannot be proven in PVS itself. In particular, Proposition 12 has to be proven for each particular instance of e and $[e]_n^\Gamma$. This is not a major drawback as, in addition to `numerical`, we have developed a strategy called `sharp` that discharges the sequent $\Gamma \vdash e \in [e]_n^\Gamma$ whenever is needed. We assume that PVS strategies are conservative in the sense that they do not add inconsistencies to the theorem prover. Therefore, if `numerical` succeeds to discharge a particular goal the answer is correct.

5.3 Decorrelation

Decorrelation is a well-known problem in interval arithmetic. Daumas et al. describe in [5] additional strategies in the Interval package that address this problem. Those strategies, which are intended for verification of numerical algorithms, are computationally intensive and not suitable for interactive theorem proving. In contrast, the strategy `numerical` uses two basic methods to reduce decorrelation. First, it automatically rearranges arithmetic expressions using a simple factorization algorithm. Due to the sub-distributivity property, factorized interval expressions are likely to be more accurate than non-factorized ones. Second, a configurable parameter allows the user to specify a splitting parameter for interval sub-divisions. This technique is described in detail in [5]. The

naïve implementation of interval sub-divisions in `numerical` is exponential with respect to the number of interval variables.

A set of lemmas of the NASA Langley PVS Libraries are also used as rewriting rules on arithmetic expressions prior to numerical evaluations. This set of lemmas is parameterizable and can be extended by the user. For instance, trigonometric functions applied to notable angles are automatically rewritten to their exact value. Although is not currently implemented, this approach can also be used to normalize angles to the range $-\pi \ldots \pi$ that is suitable for the trigonometric bounding functions in Sections 3.3.

5.4 Numerical Evaluations

To avoid symbolic evaluations, `numerical` is implemented using computational reflection [11, 2, 26]. Interval expressions are translated to Common Lisp (the implementation language of PVS) and evaluated there. The extraction and evaluation mechanism is provided by the PVS ground evaluator [23]. The result of the evaluation is translated back to the PVS theorem prover using the PVSio package developed by one of the authors [17].

We illustrate the use of the `numerical` with some examples. Lemma `tr35` is the PVS version of Formula (1). The proof is just one step of `numerical` with no parameters:

```
g : posreal = 98/10          %[m/s^2]
v : posreal = 250×514/1000 %[m/s]

tr35: LEMMA 3×π/180 ≤ g×tan(35×π/180)/v
%|- tr35: PROOF (numerical) QED
```

Another example is the proof of the inequality 4.1.35 in Abramowitz & Stegun [1]:

$$\forall x \ : \ 0 < x \leq 0.5828 \ \Rightarrow \ |\ln(1-x)| < \frac{3x}{2}.$$

The key to prove this inequality is to prove that the function

$$G(x) = \frac{3x}{2} - \ln(1-x)$$

satisfies $G(0.5828) > 0$. In PVS:

```
G(x|x < 1): real = 3×x/2 - ln(1-x)

A_and_S : lemma
  let x = 5828/10000 in
    G(x) > 0
%|- A_and_S : PROOF (numerical :defs "G") QED
```

In this case, the optional parameter `:defs "G"` tells `numerical` that the user-defined function `G` has to be considered when performing the numerical evaluation. The original proof of this lemma in PVS required the manual expansion of 19 terms of the ln series.

6 Conclusion

We have presented a pragmatic and safe way to perform ordinary real number computations in formal proofs. To this end, bounds for non-algebraic functions were established based on provable properties of their approximation series. Furthermore, a package for interval arithmetic was developed. The package includes a strategy that automatically discharges numerical inequalities and interval inclusions.

The PVS Interval package, which is available at `http://research.nianet.org/~munoz/Interval`, contains in total 306 lemmas. It is roughly 10 thousand lines of specification and proofs and 1 thousand lines of strategy definitions. These numbers do not take into account the bounding functions which have been fully integrated to the NASA Langley PVS Libraries (`http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`). It is difficult to estimate the human effort for this development as it has evolved over the years from an original axiomatic specification to a fully foundational set of theories. As far as we know, this is the most complete formalization of exact real arithmetic and interval arithmetic within a theorem prover.

Research on interval analysis and exact arithmetic is rich and abundant (see for example [14, 16, 10]). The goal of interval analysis is to compute an upper bound of the round-off error in a computation performed using floating-point numbers. In contrast, in an exact arithmetic framework, an accuracy is specified at the beginning of the computation and the computation is performed in such way that the final result respects this accuracy.

Real numbers and exact arithmetic is also a subject of increasing interest in the theorem proving community. Pioneers in this area were Harrison and Gamboa who, independently, developed extensive formalizations of real numbers for HOL [12] and ACL2 [8]. In Coq, an axiomatic definition of reals is given in [15], and constructive definitions of reals are provided in [3] and [19]. As real numbers are built-in in PVS, there is not much meta-theoretical work on real numbers. However, a PVS library of real analysis was originally developed by Dutertre [6] and currently being maintained and extended as part of the NASA Langley PVS Libraries. An alternative real analysis library is proposed in [9].

Closer to our approach are the tools presented in [4] and [5]. These tools generate bounds on the round-off errors of numerical programs, and formal proofs that these bounds are correct. The formal proofs are proof scripts that can be checked off-line using a proof assistant.

Our approach is different from previous works in that we focus on automation and pragmatism rather than accuracy. In simple words, our practical contribution is a symbolic pocket calculator for real number computations in formal proofs.[2] Thanks to all the previous developments in theorem proving and real numbers, lemmas like Lemma `tr35` and Lemma `A_and_S` are provable in HOL, ACL2, Coq, or PVS. The Interval package and, in particular, the strategy `numerical` make these proofs routine in PVS.

[2] The results are – of course – only as sound as PVS.

Acknowledgment. The authors would like to thank Marc Daumas for his early interest on this work, and his key suggestion to use interval arithmetic to automate our initial lower/upper bound technique. We are also grateful to Jacques Fleuriot for providing the original proofs of trigonometric bounds in Isabelle/HOL, to Hanne Gottliebsen for providing the same proofs in PVS, to Jeff Maddalon for comments on early drafts of this manuscript, and to the anonymous referees for their insightful comments that helped to improve this document. This work was supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NCC-1-02043.

References

1. M. Abramovitz and I. Stegun. *Handbook of Mathematical Functions*. National Bureau of Standards, 1972.
2. S. Boutin. Using reflection to build efficient and certified decision procedures. *Lecture Notes in Computer Science*, 1281:515–530, 1997.
3. A. Ciaffaglione. *Certified Reasoning on Real Numbers and Objects in Coinductive Type Theory*. PhD thesis, Università degli Studi di Udine, 1993.
4. M. Daumas and G. Melquiond. Generating formally certified bounds on values and round-off errors. In *Real Numbers and Computers*, pages 55–70, Dagstuhl, Germany, 2004.
5. M. Daumas, G. Melquiond, and C. Muñoz. Guaranteed proofs using interval arithmetic. Accepted for publication at the 17th IEEE Symposium on Computer Arithmetic, 1995.
6. B. Dutertre. Elements of mathematical analysis in PVS. In J. von Wright, J. Grundy, , and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference. TPHOLs'97*, number 1125 in Lecture Notes in Computer Science, pages 141–156, Turku, Finland, August 1996. Springer-Verlag.
7. J. Fleuriot and L. Paulson. Mechanizing nonstandard real analysis. *LMS Journal of Computation and Mathematics*, 3:140–190, 2000.
8. R. Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2*. PhD thesis, University of Texas at Austin, May 1999.
9. H. Gottliebsen. *Automated Theorem Proving for Mathematics: Real Analysis in PVS*. PhD thesis, University of St Andrews, June 2001.
10. P. Gowland and D. Lester. A survey of exact computer arithmetic. In Jens Blanck, Vasco Brattka, Peter Hertling, and Klaus Weihrauch, editors, *Computability and Complexity in Analysis*, volume 272 of *Informatik Berichte*, pages 99–115. FernUniversität Hagen, September 2000. CCA2000 Workshop, Swansea, Wales, September 17–19, 2000.
11. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.
12. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.
13. T. Hickey, Q. Ju, and M.H. van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 2001.
14. R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996.
15. M. Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, décembre 2001.

16. V. Ménissier. *Arithmétique Exacte : Conception, Algorithmique et Performances d'une Implantation Informatique en Précision Arbitraire*. PhD thesis, Université Paris VI, Paris, France, 1994.
17. C. Muñoz. PVSio reference manual – Version 2.a. Available from `http://research.nianet.org/~munoz/PVSio`, 2004.
18. C. Muñoz, V. Carreño, G. Dowek, and R.W. Butler. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer*, 4(3):371–380, 2003.
19. M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. PhD thesis, Radboud University Nijmegen, 1994.
20. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
21. M. Pour-El and J. Richards. *Computability in Analysis and Physics*. Perspectives in Mathematical Logic. Springer, Berlin, 1989.
22. R.M. Robinson. Review of "Peter, R., Rekursive Funktionen". *The Journal of Symbolic Logic*, 16:280–282, 1951.
23. N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at `http://www.csl.sri.com/shankar/PVSeval.ps.gz`.
24. E. Specker. Nicht konstruktiv beweisbare Sätze der Analysis. *The Journal of Symbolic Logic*, 14(3):145–158, 1949.
25. A. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.
26. F. W. von Henke, S. Pfab, H. Pfeifer, and H. Rueß. Case Studies in Meta-Level Theorem Proving. In Jim Grundy and Malcolm Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1479 in Lecture Notes in Computer Science, pages 461–478. Springer-Verlag, September 1998.
27. A. Yakovlev. Classification approach to programming of localizational (interval) computations. *Interval Computations*, 1(3):61–84, 1992.

Verifying a Secure Information Flow Analyzer

David A. Naumann*

Stevens Institute of Technology, Hoboken NJ 07030, USA
naumann@cs.stevens.edu

Abstract. Denotational semantics for a substantial fragment of Java is formalized by deep embedding in PVS, making extensive use of dependent types. A static analyzer for secure information flow for this language is proved correct, that is, it enforces noninterference.

1 Introduction

This paper reports on the use of the PVS theorem prover [13] to formalize a semantic model for a core fragment of Java and machine check the correctness of a secure information flow analyzer.

The primary objective of the project is to check the results of Banerjee and Naumann [2]. Their work specifies a static analysis, in the form of a type system, for secure information flow in a substantial fragment of Java including mutable state, recursive types, dynamic allocation, inheritance and dynamic binding, type casts, and the code-based access control mechanism (stack inspection [6]). Security policy, i.e., confidentiality/integrity, is expressed by annotating object fields and method signatures with levels from a security lattice. The main result of [2] is that the typing rules are sound in the sense that a program deemed safe by the analysis is noninterferent, w.r.t. the policy (as in [18], see the survey by Myers and Sabelfeld [15]). A detailed proof of soundness has undergone peer review but it is sufficiently complicated to merit machine checking.

In recent years, operational semantics has been popular but [2] is based on a denotational model in the style of Scott and Strachey. In particular, the interesting semantic domains are higher order dependent function spaces and the main results are proved by fixpoint induction. A second objective of our project is to explore how well PVS supports formalizing and reasoning about such a model. Owing to the use of a denotational model we reap the benefits of a deep embedding while avoiding the need to manipulate or even define syntactic substitution. Moreover there is no explicit method call stack, which simplifies reasoning about noninterference.

A third objective was for the author to gain his first experience with a proof assistant. To give this a more scholarly spin, let us say the objective was to assess the effectiveness of PVS as a proof assistant for a user without expertise in mechanical theorem proving.

The project is largely completed. With the omission of code-based access control, the results of [2] have been completely formalized and proved. The omission was made only to scale down this first phase of the project. The static analysis is presented as

* Supported in part by NSF CCR-0208984, CCF-0429894, and CCR-ITR-0326540; ONR N00014-01-1-0837; and the New Jersey Commission on Science and Technology.

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 211–226, 2005.

a recursive function that implements the typing rules of [2], and an arbitrary security lattice is considered, whereas the paper considers only the two-point lattice.

The design decision with the most far-reaching consequences was to use higher order dependent types to encode the semantics in a way that closely matches the paper [2]. The PVS type system is undecidable; the type checker generates type correctness conditions (TCCs) that must be proved. Once proved they are useful as lemmas. The use of dependent predicate types to express, e.g., the absence of dangling pointers and ill-typed program states, means that type-correctness of our definitions requires nontrivial proof in some cases. A tangential consequence is type soundness for the modeled language.

PVS is notable for its high level of automation and integrated decision procedures but to our knowledge there are no previous large applications involving higher order dependent types like ours. A considerable amount of user interaction appears to be needed for the proofs, for reasons explained in the sequel. The proofs mostly follow the structure of those in the paper, as reflected in the formulation of lemmas. But rather than transcribing the details from the paper, I tried to carry out proofs by the seat of my pants and I found PVS to be a fairly pleasant assistant in this working style. As a beginning user I refrained from trying to do sophisticated control of rewriting, or defining new proof strategies, or using automatic conversions between total and partial functions represented by lifting. I decided not to factor large proofs by introducing lots of little lemmas with no independent interest, given that PVS offers a graphical display that facilitates inspection of sequents at intermediate points. Some proof scripts run to several dozen lines and a few are a hundred lines.

Axioms are used only to express assumptions as follows: (a) the program to be analyzed is syntactically well formed (e.g., ordinary type correctness, subclassing is acyclic); (b) flow policy is well formed, i.e., invariant with respect to subclassing; and (c) the memory allocator, otherwise arbitrary, returns fresh locations. The axioms are simple and soundness is obvious. PVS theory interpretations have been used to confirm this claim but that is omitted for lack of space.

Related work. Strecker [16] uses Isabelle/HOL to show soundness for a language and security typing rules very similar to those in [1], i.e., a sequential fragment of Java, without access control. Strecker's result, like that in [1], makes an assumption of "parametricity" for the memory allocator in order to use equality rather than an arbitrary bijection on locations in the definition of noninterference. This simplifies the proofs considerably but is at odds with memory management in practice. Strecker also confines attention to the two-element lattice. The present work confirms the wisdom of those simplifications. Although an arbitrary lattice is needed for practical applications, it results in an additional quantifier in each of the key properties (indistinguishability, write-confinement, and noninterference) and this pervades the proofs. Treating memory allocation realistically necessitates the use of a bijection on locations in the definition of indistinguishability. As Strecker remarks, this means that there are fewer opportunities to exploit provers' built-in equality reasoning.

Jacobs, Pieters and Warnier [8] report on using PVS to verify soundness of a static information flow analysis for a simple imperative language. The analysis is based on abstract interpretation instead of types, which allows a variable to be used for information of different levels at different times. Hähnle and Sands [5] use the KeY tool, an

interactive theorem prover, to prove confidentiality of programs in a subset of JavaCard. Confidentiality is formalized in dynamic logic, following the semantic approach of Joshi and Leino [9]. Rushby [14] uses EHDM to check an "unwinding theorem" that says the "no read up" and "no write down" rules [15] for individual steps suffice to ensure noninterference. This elegant theory is set in an abstract model rather than a programming language; the focus is on intransitive noninterference.

Overview. This paper reviews the work of [2] and describes the formalization in PVS of the semantics (Sect. 2), policy and static analysis (Sect. 3), and proof of noninterference (Sect. 4). Sect. 5 concludes. For readers not familiar with PVS, key notations are briefly explained. More details can be found in a technical report on the author's home page, together with the PVS files.

2 Formalizing of the Language: Syntax and Semantics

This section describes the object language, as defined "in the paper" (meaning [2]), and its formalization in PVS. For clarity, some minor details of the description differ from the paper and the PVS code, but only in ways that seem unlikely to lead to confusion.

Signature of a class table. A well formed program is given in the form of a class table, i.e., a collection of class declarations, each giving a superclass, field types, method signatures, and method implementations. The paper follows [7] in using several auxiliary functions: $superC$ gives the direct superclass declared by a class named C; $dfieldsC$ gives the field declarations $f0 : T0, f1 : T1, \ldots$ of C; $fieldsC$ combines the declared and inherited field declarations; and $mtype(m,C)$ gives the parameter and return type for method m declared or inherited in C. If there is no such method, $mtype(m,C)$ is undefined. Fields and methods of a class may make mutually recursive reference to any other classes. We assume there are given, disjoint sets of field names, class names, and method names, ranged over by f, C, and m respectively. Finally, $mbody(m,C)$ gives the method body, if method m has a declaration in class C, and is undefined otherwise.

The data types T in Java consist of primitive types (such as `boolean` and `int`), names of declared classes, and the unit type (`void`). Data types are the types of fields, local variables, method parameters and method return; the unit type is used as return type for a method called only for its effect on state. Data types are given by the grammar

$$T ::= \texttt{bool} \mid \texttt{unit} \mid C$$

where C ranges over the set of declared class names. This is formalized as a PVS inductive datatype, `dty`, in theory `dty` which is parameterized on a nonempty set `Classname` used as argument for the constructor `classT(name: Classname)`. Datatype recognizers are named according to the usual convention, e.g., `classT?`.

Theory `classtableSig` formalizes the signature of a well formed class table, i.e., fields, superclass, and method signatures for every class. It declares uninterpreted sets `Classname`, `Methname`, and `Varname` (the latter for fields and also local variables and parameters). Unlike the paper, we do not formalize the syntax of class declarations but rather work directly with the auxiliary functions `super`, `fields`, etc.; e.g., the following uninterpreted function declaration gives method signatures:

```
mtype: [Classname, Methname
        -> lift[ [# parN: Varname, parT: dty, resT: dty #] ] ]
```

Undefinedness is represented using the lift constructor, as PVS is a logic of total functions (`[T->U]` is notation for functions, `[# lab:T #]` for records, and `r`lab` for field selection). If `mtype(m,C)` is defined (not `bottom`) then its value is a record with field `parT` giving the parameter type, `resT` giving the result type, and `parN` giving the parameter name. As in the paper, the parameter name is treated as part of the type; this loses no generality and simplifies definitions related to method call.

Method types in Java are invariant, i.e., if `C` is a subclass of `D` and inherits or overrides a method `m` of `C` then the method signatures are the same. This is expressed by

```
inherit_meths: AXIOM C <= D & up?(mtype(D,m)) => mtype(C,m) = mtype(D,m)
```

Axiom `inherit_meths` also embodies inheritance in the sense that if `m` is defined in a class then it is also defined in subclasses thereof. The set of methods defined for a given class is lifted to the level of types as follows:

```
definedMeth(C)(m): bool = up?(mtype(C,m))
DefinedMeth(C): TYPE = (definedMeth(C))
```

The first line defines a predicate and the second line uses the PVS notation of enclosing parentheses to lift a predicate to a type, here the type of methods either declared or inherited in class `C`. The declaration and inheritance of fields is treated similarly.

The declaration-based subclassing relation turned out to be slightly intricate to formalize. Theory `classtableSig` declares an uninterpreted function `super` to designate the immediate superclass (except for mapping the distinguished class `Object` to itself).

```
super: [Classname -> Classname]
super_fin_top: AXIOM EXISTS j: j > 0 & iterate(super,j)(C) = Object
```

Here `C` ranges over class names owing to the declarations

```
C, D, E:   VAR Classname          T, T1, T2: VAR dty
```

In the sequel we omit such declarations. Subclassing on class names is defined by

```
<=(C, D): bool = EXISTS j:  iterate(super,j)(C) = D
```

and this extends easily to a relation `<=` on data types. Lemmas state that `Object` is the top element and that `<=` is a preorder. The proofs require minimal user guidance, e.g., here is the script for transitivity of `<=` on class names:

```
(skosimp) (expand "<=") (lemma "iterate_add_applied[Classname]") (grind)
```

It uses a lemma about the `iterate` function from the PVS prelude as well as the most powerful strategy, `grind`, which repeatedly applies simplification, instantiation, skolemization and if-lifting.

Many of the results are proved using a secondary induction on inheritance chains, formalized using the following (where `/=` is disequality).

```
stepsToObj(C)(j): bool  =  iterate(super,j)(C) = Object
cdepth(C): nat = min(stepsToObj(C))
cdepth_super: LEMMA C /= Object => cdepth(super(C)) < cdepth(C)
```

Theory `classtableSig` includes a number of additional definitions and results concerning subclassing and inheritance. It also defines contexts for use in typing rules. In the paper, a (variable) context Γ is a partial function from variable names to data types. It is formalized in PVS as a dependent record type.

```
Vxt: TYPE = [# dom: set[Varname], map: [(dom) -> dty] #]
```

Here `(dom)` is notation for the lift of predicate `dom` to the level of types. The type `set[Varname]` is syntactic sugar for the function type `[Varname -> bool]`.

In the semantics, the state on which a command operates includes a type correct assignment of values to variables (i.e., parameters and locals including `self`) and the state of an object in the heap is a type correct assignment of values to field names. The context for fields is defined as follows, using set comprehension notation `{ f |...}`.

```
fieldVxt(C: Classname): Vxt =
  (# `dom := { f | fields(C)(f) }, `map := lambda f: ftype(C,f) #)
```

Similarly, `methParVxt(C)(m)` declares `self:C` and also the method's parameter.

Typing of method bodies. The context free syntax of expressions and commands is given by PVS datatypes `exp` and `com` in a theory `lang` which is parameterized on the sets `Classname`, `Varname`, and `Methname`. An assignment $x := e$ is represented by the term `assign(x,e,T)` with an explicit trace of the type of `e` to simplify type checking; similarly for the other constructs. The expressions are: variables, boolean literals, null, equality test, field access, type test, and type cast. The commands are: assignment, field update, new, dynamically dispatched method call, sequence and conditional. (Loops are omitted since general recursion is included.)

Theory `typing` gives the typing rules for expressions and commands. The typing judgement $\Gamma \vdash e : T$ is expressed by a predicate `expOK(V,T)(e)` where `V` ranges over contexts, `T` over data types, and `e` over expressions.

```
expOK(V,T)(e): RECURSIVE bool =
 CASES e OF
 vblV(n):  V`dom(n) & T = V`map(n) ,
 fieldAccess(e1,T1,f):expOK(V,T1)(e1) & classT?(T1) & ftype(name(T1),f)=T
... MEASURE e by <<
```

This uses pattern matching on the datatype `exp` of expressions. A variable named `n` has type `T` in context `V` if `n` is in the domain of `V` and is assigned type `T`. Recursive definitions in PVS must be proved total using an explicitly designated measure, in this case the subterm relation `<<` generated automatically by the datatype definition for `exp`.

For commands, the paper's judgement $\Gamma \vdash S$ is formalized as follows.

```
comOK(V)(S): RECURSIVE bool =
  CASES S OF
   assign(n,e,T): n /= Self & V`dom(n) & expOK(V,T)(e) & T <= V`map(n)...
```

The target `n` of assignment cannot be `self` and must be declared. The expression must have the type recorded `T` in the syntax and `T` must be a subtype of the type of `n`. The typing rules here and in the paper are syntax directed, so the semantics can be defined

by recursion on syntax. Subsumption is not present as a separate rule but is instead embodied by subtyping conditions.

All of the TCCs in theories `typing` and `classtableSig` are proved by the default strategy without user intervention.

Theory `wellformedCT` declares an uninterpreted function for method bodies.

```
mbody: [C: Classname, m: Methname ->
           lift[[# localvar: Varname, localvarType: dty, body: com #]] ]
```

To simplify the formalization slightly, each method is assumed to have exactly one local variable. The formalization also enforces that the local variable name is distinct from the distinguished names `self` and `result` as well as from the parameter name (not shown here).

Every method should have an implementation and every method declaration should be typable; this assumption is expressed as follows.

```
declaredMeth(C)(m): bool = definedMeth(C)(m) & up?(mbody(C,m))
DeclaredMeth(C): TYPE = (declaredMeth(C))
every_meth_has_body: AXIOM  definedMeth(C)(m) & NOT declaredMeth(C)(m)
                              => C /= Object & definedMeth(super(C))(m)
all_bodies_typable: AXIOM
  declaredMeth(C)(m) => comOK( bodyVxtFor(C,m) )( down(mbody(C,m))`body )
```

Semantic domains. According to the paper, the state of a method in execution is comprised of a *heap h*, which is a finite partial function from locations to object states, and a *store r* which assigns locations and primitive values to local variables and parameters. States are self-contained in the sense that all locations in fields and in variables are in the domain of the heap. For locations, we assume that a set *Loc* is given, along with a distinguished entity *nil* not in *Loc*. To track an object's class we assume given a function $loctype : Loc \to Classname$ such that there are infinitely many locations ℓ with $loctype\,\ell = C$, for each C. In the formalization, it suffices to assume that the allocator is a total function. Theory `semanticDomains` begins with the uninterpreted declarations `Loc: TYPE+` and `loctype: [Loc -> Classname]`. It imports theory `value[Loc]` which defines a datatype of values with constructors `semNil`, `semLoc(valL: Loc)`, etc. These are classified as follows.

```
locsBelow(C)(l: Loc): bool = loctype(l) <= C
LocsBelow(C): TYPE = (locsBelow(C))
valOfType(T)(v: Value): bool =
 CASES T OF
 unitT:     semIt?(v) ,
    boolT:     semBool?(v) ,
 classT(C): semNil?(v) OR ( semLoc?(v) & locsBelow(C)(valL(v)) ) ENDCASES
ValOfType(T): TYPE = (valOfType(T))
val_subsumptionT: LEMMA  T <= T1 & valOfType(T)(v) => valOfType(T1)(v)
```

A number of subsumption properties are needed in the semantics. Formulation is a little delicate, e.g., in a case like `val_subsumptionT` this very property is needed to discharge a TCC for `valOfType(T1)(v)`. Once the TCC has been proved, one can

Table 1. Semantic domains, named by categories θ including *MEnv* for method environments; $(C, \bar{x}, \bar{T} \to T)$ for methods of C with parameters $\bar{x} : \bar{T}$, return T; and $Heap \otimes \Gamma$ for closed states.

$$\theta ::= T \mid \Gamma \mid objstate\,C \mid Heap \mid Heap \otimes \Gamma \mid Heap \otimes T \mid \theta_\perp \mid (C, \bar{x}, \bar{T} \to T) \mid MEnv$$

$$[\![\mathbf{bool}]\!] = \{true, false\} \qquad [\![\mathbf{unit}]\!] = \{it\} \qquad [\![C]\!] = \{nil\} \cup \{\ell \mid \ell \in Loc \wedge loctype\,\ell \leq C\}$$

$$[\![\Gamma]\!] = \{r \mid dom\,r = dom\,\Gamma \wedge r\,\mathsf{self} \neq nil \wedge \forall x \in dom\,r \bullet r\,x \in [\![\Gamma\,x]\!]\}$$

$$[\![objstate\,C]\!] = \{s \mid dom\,s = dom(fields\,C) \wedge \forall (f : T) \in fields\,C \bullet s\,f \in [\![T]\!]\}$$

$$[\![Heap]\!] = \{h \mid dom\,h \subseteq_{fin} Loc \wedge closed\,h \wedge \forall \ell \in dom\,h \bullet h\ell \in [\![objstate\,(loctype\,\ell)]\!]\}$$
where $closed\,h$ iff $rng\,s \cap Loc \subseteq dom\,h$ for all $s \in rng\,h$

$$[\![Heap \otimes \Gamma]\!] = \{(h, r) \mid h \in [\![Heap]\!] \wedge r \in [\![\Gamma]\!] \wedge rng\,r \cap Loc \subseteq dom\,h\}$$

$$[\![Heap \otimes T]\!] = \{(h, v) \mid h \in [\![Heap]\!] \wedge v \in [\![T]\!] \wedge (v \in Loc \Rightarrow v \in dom\,h)\}$$

$$[\![C, \bar{x}, \bar{T} \to T]\!] = [\![Heap \otimes (\bar{x} : \bar{T}, \mathsf{self} : C)]\!] \to [\![(Heap \otimes T)_\perp]\!]$$

$$[\![MEnv]\!] = \{\mu \mid \forall C, m \bullet \mu Cm \text{ is defined iff } mtype(m, C) \text{ is defined, and } \mu Cm \in [\![C, pars(m, C), mtype(m, C)]\!] \text{ if } \mu Cm \text{ defined } \}$$

appeal to it to get an immediate proof of the lemma. But the lemma needs to be explicitly stated in order to generate the TCC.

In the paper, notation for the hierarchy of semantic domains is based on syntactic categories θ, see Table 1. These categories are not separately formalized in PVS. Instead there are just named types, e.g., for $[\![\Gamma]\!]$ we define `Store(V)`. (Recall that identifier `V` is used instead of Γ.) A store for context `V` maps each name `n` in the domain of `V` to a value of the type assigned to `n` in `V`.

```
Store(V): TYPE = [n: (V`dom) -> ValOfType(V`map(n))]
```

Heaps are defined in two stages, the second imposing the containment condition.

```
ObjState(C): TYPE = Store(fieldVxt(C))
preHeap: TYPE = [# dom: finite_set[Loc],
                   map: [ l:(dom) -> ObjState(loctype(l)) ] #]
closedStore(V)(h: preHeap)(r: Store(V)): bool =
  FORALL (n:(V`dom)):
   classT?(V`map(n)) & NOT semNil?(r(n)) => h`dom(valL(r(n)))
heap(h: preHeap): bool =
  FORALL ( l:(h`dom) ):
   LET V = fieldVxt(loctype(l)), r = h`map(l) IN closedStore(V)(h)(r)
Heap: TYPE = (heap)
state(V)(h: Heap, r: Store(V)): bool = closedStore(V,h)(r)
State(V): TYPE = (state(V))
```

If `s: State(V)` then `s`1` is the heap part, using the PVS projection notation.

In the paper, no domains are explicitly defined for expressions but it is stated that the meaning of an expression $\Gamma \vdash e : T$ is a function $[\![Heap \otimes \Gamma]\!] \to [\![T_\perp]\!]$ that takes a state $(h, r) \in [\![Heap \otimes \Gamma]\!]$ and returns either a value $v \in [\![T]\!]$, such that $(h, v) \in [\![Heap \otimes T]\!]$ (i.e., v is in $dom\,h$ if v is a location), or the improper value $\perp$ which represents errors. In the formalization the domain for expressions is made explicit:

```
preExpr(V,T): TYPE = [ State(V) -> lift[ValOfType(T)] ]
contValOfType(T, h)( v ): bool =
  valOfType(T)(v) & ( classT?(T) & NOT semNil?(v) => h‘dom(valL(v)) )
CValOfType(T: dty, h: Heap): TYPE = (contValOfType(T,h))
semExpr(V,T)(g: preExpr(V,T)): bool =
  FORALL (s:State(V)): liftContValOfType(T,s‘1)(g(s))
SemExpr(V,T): TYPE = (semExpr(V,T))
```

The paper says that a typable command $\Gamma \vdash S$ denotes a function $[\![MEnv]\!] \to [\![Heap \otimes \Gamma]\!] \to [\![(Heap \otimes \Gamma)_\perp]\!]$. In proving noninterference, which involves extending bijections on the heap domain, it became apparent that commands must have an additional property not made clear in the paper: Any location allocated in the initial heap is still allocated in the final heap. (By not modeling garbage collection, we simplify the formulation of noninterference.) This condition could be proved as a lemma but is instead imposed on the semantic domains for commands (and also method meanings). This is formalized as follows.

```
preSemCommand(V): TYPE = [ State(V) -> lift[State(V)] ]
semCommand(V)(g: preSemCommand(V)): bool =
  FORALL (s: State(V), l:(s‘1‘dom)):  up?(g(s)) => down(g(s))‘1‘dom(l)
SemCommand(V): TYPE = (semCommand(V))
```

Finally, the denotation of a method `m` takes a pair `(h,r)`, where `r` is a store with the arguments, i.e., `self` and for the parameter. (An alternate formulation would pass a tuple of values, with the benefit that the parameter name would not be significant in method types. The cost would be additional definitions of indistinguishability etc. for tuples.) It returns $\perp$ or a pair `(h1,v)` where `v` is the result value and `h1` the updated heap. The semantic domain `SemMeth(C,m)` imposes the conditions that `v` is in the domain of `h1` and that the domain of `h1` extends that of `h` (the latter condition is missing from the equation for $[\![(C, \bar{x}, \bar{T} \to T)]\!]$ in Table 1).

Method environment. The semantics for commands, discussed later, is defined in terms of a method environment which provides for each defined method an appropriate semantics.

```
MethEnv: TYPE = [C: Classname -> [m: DefinedMeth(C) -> SemMeth(C,m)]]
```

Theory `semCT` defines the semantics of a class table as a method environment obtained as the limit of a chain of approximations.

```
approxMethEnv(j)(C): RECURSIVE [m:DefinedMeth(C) -> SemMeth(C,m)] =
  lambda (m:DefinedMeth(C)):
     IF j=0 THEN abortMeth(C,m)
     ELSE IF declaredMeth(C)(m) THEN mbodySem(C,approxMethEnv(j-1))(m)
     ELSE restrict[...]( approxMethEnv(j)(super(C))(m) ) ENDIF ENDIF
   MEASURE lex2(j, cdepth(C)) BY <
```

For the 0th approximation, every method aborts. For the jth approximation, if m is declared in C then its semantics is given in terms of the semantics of its body, interpreted with respect to the $(j-1)$st approximation for methods it calls. (We defer the definition of `mbodySem`.) This exactly mirrors an operational semantics in which the call stack

size is bounded by $j-1$ (note that there is no stack in the denotational model). The semantics of a complete program has a one-line definition:

```
semCT: MethEnv = lub( approxMethEnv )
```

This uses the least upper bound operator defined in theory `orderDomains`, where it is shown that the semantic constructs are monotonic and continuous. Operationally, taking the `lub` removes the bound on stack size. We omit further details.[1]

Returning to `approxMethEnv`, if *m* is inherited then its semantics is inherited, i.e., given by `approxMethEnv(j)(super(C))(m)`. But this function is applied to a smaller domain (the argument stores where `self` has type `C` rather than `super(C)`), whence the need for `restrict[...]` to make the conversion explicit for the PVS type checker.

The definition of `approxMethEnv` generates eight TCCs. One is that the given order is well founded. Three of the TCCs require more than one step of user interaction, mainly to show that `approxMethEnv(j)(super(C))(m)`, which has type `SemMeth(super(C),m)`, also has type `SemMeth(C,m)` (when its domain is restricted `State(methParVxt(C)(m))`). The proofs use axiom `every_meth_has_body`, lemma `cdepth_super`, and the following.

```
inherit_defined: LEMMA
   FORALL C, (m: DefinedMeth(super(C))), (g: SemMeth(super(C),m)):
     semMeth(C, m)(restrict[...](g))
```

For lemma `inherit_defined` the proof is easy:

```
(skosimp*)(typepred "g!1")(use "super_above")(use "semMeth_subsumpt")(prop)
```

On the other hand, to discharge the subtyping TCCs generated by `inherit_defined` requires several steps, in two of which instantiations needed to be supplied. The TCCs for this lemma are then used subsequent proofs.

Semantics of commands and method bodies. Theory `comSemantics` gives the semantics for commands, given an uninterpreted function for memory allocation.

```
fresh: [Classname, Heap -> Loc]
freshness: AXIOM  loctype(fresh(C,h)) = C & NOT h‘dom(fresh(C,h))
```

A simple and realistic model for this assumption is to let *Loc* be the set of pairs (i,C) with *C* a classname and *i* a natural; *fresh* chooses the least unused *i*.

For each syntactic command construct we explicitly define a semantic operation, e.g., here is the semantic operation that allocates a fresh object and assigns it to `n`:

```
newS(V)(n: {n | V‘dom(n) & n/=Self}, C: {C | below(V‘map(n))(classT(C))})
       (s: State(V)): lift[State(V)] =
  LET l = fresh(C, s‘1),
      h1 = (# ‘dom := add(l, s‘1‘dom),
              ‘map := s‘1‘map WITH  [ (l) |-> initObState(C)] #)
  IN up( updateVar(V)(n, (h1, s‘2), semLoc(l)) )
```

[1] The `lub` of method environments admits an elementary characterization, owing to the simple concrete representation for states. If method meanings were stored in the heap then we would need to work with the solution of a nontrivial domain equation. Indeed, Levy [12] gives a denotational model for a higher order language with pointers, but the model does not capture relational parametricity or recursive types.

The semantic function comSem has a succinct definition, mapping each construct to its corresponding semantic operation. In the paper, the semantics $[\![\Gamma \vdash S]\!]$ is only defined for derivable judgement $\Gamma \vdash S$. A direct encoding of this would lift the predicate for typing, comOK(V), to a type, but then we would have to state and prove an induction rule for the set of typable commands. In order to use the induction rule generated by PVS for the datatype com, i.e. for proofs to begin simply (induct "S"), we define semantics for all commands, typable or not.

```
comSem(V: Vxt, me: MethEnv)(S: com): RECURSIVE SemCommand(V) =
  IF NOT comOK(V)(S) THEN abortCom(V) ELSE
  CASES S OF
    new(n, C):        newS(V)(n, C) ,
    seq(S1,S2):       seqS(V)( comSem(V,me)(S1), comSem(V,me)(S2) ) ,
    mcall(n,e1,m,e2,T1,T2):   % that is, n := e1.m(e2)
       mcallS(V,me)( n, T1, expSem(V,T1)(e1), m, expSem(V,T2)(e2) ) ,
 ...MEASURE S by <<
```

Type soundness. The theories discussed so far give a deep embedding of the syntax and semantics of a fragment of Java with features including mutable objects, recursive class definitions, type casts, pointer equality, and inheritance. The semantic domains express invariants like these: self is not null; objects are never deallocated; the value of any field of any object is an element of the (denotation of) the type declared for the field. The net effect is that type soundness is a consequence of definedness of the semantics. To see how this works, consider an assignment assign(n,e,T) typable in context V. The semantics is assignS(V)(n, T, expSem(V,T)(e)), using the following.

```
assignS(V)(n: {n|V`dom(n) & n/=Self}, T:Below(V`map(n)), g:SemExpr(V,T))
          ( s: State(V) ): lift[State(V)] =
   IF up?(g(s)) THEN up( updateVar(V)(n,s,down(g(s))) ) ELSE bottom ENDIF
```

Note that assignS(V) is applicable only to a variable n in the domain of V and not equal to self; moreover, the type T of the assigned expression must be a subtype of the type of n in V. The declaration indicates that assignS(V)(n,T,g) applies to a state for V and returns a lifted state. An additional result says that in fact it has the type SemCommand(V).

```
assignS_type: JUDGEMENT
  assignS(V)(n:{n|V`dom(n) & n/=Self}, T:Below(V`map(n)), g:SemExpr(V,T))
  HAS_TYPE SemCommand(V)
```

This can be read as a lemma, saying that if assign(n,e,T) typechecks then it denotes a function satisfying the invariants imposed by SemCommand(V). The definition of comSem generates three TCCs for assign, which can be discharged easily using assignS_type and inversion of the typing rule.[2]

The domain definitions, subsumption, inheritance, etc., come together in the semantics method call, mcallS. The TCCs for this definition are among the longer proofs in the development, as user interaction is needed to invoke various subsumption lemmas.

[2] The JUDGEMENT form provides refined type information to the type checker, to reduce redundant TCCs. It is of little use in our work because it does not cater for dependent types, e.g., JUDGEMENT loc(l: locsBelow(C)) HAS_TYPE ValOfType(classT(C)) is not allowed because C is free. Instead we use LEMMA FORALL (C:Classname, l: LocsBelow(C)): valOfType(classT(C))(loc(l)).

3 Formalizing the Static Analysis and Noninterference

Security policy. Noninterference for a given security labeling means that if a pair of initial states are indistinguishable by an observer who only sees fields/variables labeled below level *L* then the corresponding final states are indistinguishable below *L*.

Theory `lattice` defines the relevant operations on an uninterpreted type, `Level`. Variables `L`, `L1` etc range over `Level`. Security policy assigns a level to each field as well as to method parameters and result. Theory `indistinguishable` defines the indistinguishability of states, with respect to a given policy.

```
methPolicy:TYPE = [Classname,Methname -> [# self, par, res, hp: Level #]]
fieldPolicy: TYPE = [Classname -> [Varname -> Level]]
```

The method policy for a given class and method gives not only levels for the parameters and result, serving as upper bounds on their information, but also a lower bound on the levels of fields written (`hp`). We refrain from tying policy to a particular class table signature, but rather let the policy assign arbitrary values for undefined methods and fields.

Indistinguishability of values involves a partial bijective renaming of locations to mask allocation effects. Theory `indistinguishable` develops suitable theory for type-respecting partial bijections on locations, e.g., how composition of such relations behaves. Indistinguishability of values of type `T`, relative to a typed bijection `rho`, is defined as follows:

```
indis( rho, T )( v, v?: ValOfType(T) ): bool =
   CASES T OF
     unitT: TRUE ,
     boolT: v = v? ,
     classT(C): ( semNil?(v) & semNil?(v?) )
             OR ( semLoc?(v) & semLoc?(v?) & rho( valL(v), valL(v?) ) )
```

Where the paper uses v, v' for related pairs, the PVS formalization uses `v?` because symbol $'$ is not allowed in identifiers. For this reason we mostly avoid the convention of using `?` to signal predicate names. The definition above induces the notions of indistinguishability (overloading the name `indis`) for each of the other semantic domains. We need the notion of a policy for stores:

```
Levels(V):   TYPE = [(V`dom) -> Level]
```

Indistinguishability of stores with respect to an observer of level `L` and local variable labeling `lev` is defined as follows:

```
indis( L, rho, V, (lev: Levels(V)) ) ( r, r?: Store(V) ): bool =
   FORALL (n:{n|V`dom(n) & lev(n)<=L}): indis(rho,V`map(n))( r(n), r?(n))
```

The definition says that `r` is indistinguishable from `r?` for an observer at level `L` provided that the values for `n` are indistinguishable for every variable `n` with level at most `L`.

Confinement. Confinement expresses, semantically, the key rules "no read down" and "no write up" [15]. For an expression meaning `g` to be *read confined* for observers at level `L` means that it cannot distinguish states that are indistinguishable for *L*.

```
readConf(L, V, T, fpField)(levs: Levels(V))(g: SemExpr(V,T)): bool =
  FORALL rho, (s, s?: State(V)):
   indis(L,rho,V,fpField,levs)(s,s?) => indis(rho,T)(g(s),g(s?))
```

For a command to be *write confined* above L means that it writes no variable or field below level L; that is, its final state is indistinguishable, for observers not at a level above L, from the initial state, using as renaming bijection the identity on the domain of the initial heap. Write confinement is also defined for methods; details omitted.

Safety. Theory safe gives the static analysis in terms of uninterpreted policy functions and also an assignment of levels to the local variable(s) of method bodies.[3]

```
mPolicy: [Classname, Methname -> [# self, par, res, hp: Level #]]
fPolicy: [Classname -> [Varname -> Level]]
localLevel: [Classname, Methname, Varname -> Level]
```

It is assumed that the policies are invariant with respect to subclassing.

```
inherit_meth_lev:AXIOM C<=D & up?(mtype(D,m))=> mPolicy(C,m)=mPolicy(D,m)
```

In the paper, the static analysis is specified using labeled types (T, κ) for expressions, where T is a data type and κ a security level. For commands, the judgement $\Delta \vdash S : \kappa, \kappa'$ expresses that S is secure, with respect to a given policy assigning levels to locals (Δ), fields, and method parameters/returns, and moreover S writes fields (resp. locals) of level at least κ (resp. κ'). There is a similar judgement for expressions. The paper gives rules to inductively define these judgements but in PVS we define a recursive function that, for expression e, gives the least L at which e would be typable in the paper. This serves as a deterministic and reasonably efficient implementation.

In the paper, the analysis, like the semantics, is specified only for typable programs. In the PVS formalization, safety is defined for all programs but only typable programs are deemed safe. (Compare comSem.)

```
expSafe(V: Vxt, lev: Levels(V), T: dty )(e: exp): RECURSIVE lift[Level] =
  If NOT expOK(V,T)(e) THEN bottom ELSE
  CASES e OF vblV(n): up( lev(n) ) ,
             fieldAccess(e1, T1, f):
              s_lub( expSafe(V,lev,T1)(e1), up(fPolicy(name(T1))(f)) )...
```

For a method declaration to be safe, mSafe, is defined to mean that its body is safe with respect to the method policy. Function comSafe is similar to expSafe. Finally, safety of a whole program is defined as follows.

```
safe_CT: bool = FORALL C, (m: DeclaredMeth(C)): mSafe(C)(m)
```

4 Main Results

Confinement. Theory confinement shows that the analyzer ensures read and write confinement. For the semantic operation used to interpret each program construct, there is a lemma expressing conditions under which the semantics is confined. For example, here is the case that the expression is a variable:

[3] The formalization could be improved by factoring out the policy from both theories safe and indistinguishable.

```
vblS_read_conf: LEMMA FORALL ( V: Vxt, levs: Levels(V) ):
   V‘dom(n) & V‘map(n) = T & levs(n)<= L
   => readConf(L, V, T, fpField)(levs)( vblS(V)(n) )
```

These lemmas are used to prove that if the analysis returns level L for an expression then its semantics is read confined at that level. Similarly, if the analyzer says a command is safe at a certain level pair LLp then it is write confined for LLp.

```
safe_com_write_conf: LEMMA
  FORALL ( V: Vxt, lev: Levels(V), S: com, me: WriteConfMenv ):
   LET LLp = comSafe(V,lev,S)
   IN up?(LLp) => writeConf(down(LLp), V, fPolicy, lev)( comSem(V,me)(S))
```

Note the antecedent that method meanings in me are write confined. The lemma is proved by induction on the structure of S; in each case one inverts the safety rule and uses lattice properties to establish the hypothesis for the corresponding semantic confinement lemma.

To discharge the assumption about method environment, we have to show that if all method bodies are safe then all their meanings are write confined. This is proved by showing that all approximations are write confined and that this implies the same for the least upper bound.

```
write_conf_approx: LEMMA
   safe_CT => (FORALL j: writeConfMenv( approxMethEnv(j) ))
write_conf_admissible: LEMMA
   FORALL ( app: AscendingMenvs ):
     (FORALL j: writeConfMenv(app(j))) => writeConfMenv( lub(app) )
safe_CT_write_conf: LEMMA  safe_CT => writeConfMenv( semCT )
```

Admissibility is a straightforward series of steps unfolding the definitions, owing to our explicit characterization of lub in theory orderDomains. For write_conf_approx the proof goes by induction on the lexical ordering of j and depth of inheritance (using lemma cdepth_super).

The least satisfying proof in this theory is the write confinement lemma for semantics of method call. It is long, mainly because a number of TCCs are generated at several points where lemmas are invoked. Some of the TCCs are discharged by a simple strategy which tries all the TCCs from the definition of the semantic function mcallS. For a number of others the strategy fails and I have to explicitly appeal to a specific one of the TCCs associated with the definition of mcallS in theory comSemantics.

Noninterference. It is only in the final theory, safeProps, that the TCCs are really onerous. First come definitions, e.g., what it means for a command to be noninterferent with respect to a policy lev for variables (the policy for fields is imported from theory safe) and an observer that can see variables and fields of level at most L:

```
nonint( L, V, (lev: Levels(V)) )( g: SemCommand(V) ): bool =
   FORALL rho, (s, s?: State(V)):
    indis(L,rho,V,fPolicy,lev)(s,s?) & up?(g(s)) & up?(g(s?))
    => EXISTS tau: subset?(rho,tau)
       & indis(L,tau,V,fPolicy,lev)(down(g(s)), down(g(s?)))
```

There is a similiar notion, `nonint(L,C,m)`, of noninterference for the meaning of method `m` at class `C`. A method environment is noninterferent provided every method meaning is noninterferent for observers at every level `L`. (The definition is factored into two to fit a standard induction rule.)

```
nonintMenv2( me, C ): bool =
  FORALL L, (m: DefinedMeth(C)):  nonint(L, C, m)( me(C)(m) )
nonintMenv( me ): bool = FORALL C: nonintMenv2(me,C)
```

Next come noninterference lemmas for the semantic operations, for example:

```
nonint_assignS: LEMMA
 FORALL L, V, (lev: Levels(V)),  (n: {n | (V`dom) & n/=Self)}),
        (T: Below(V`map(n))), (g: SemExpr(V,T)):
   readConf(lev(n),V,T,fPolicy)(lev)(g)
   => nonint(L,V,lev)( assignS(V)(n,T,g) )
```

The proof involves manipulating a pair `s,s?` of states related by `indis` and unfolding the definition of `assignS` on both `s` and `s?`. This generates two sets of TCCs. The TCCs can be discharged by using, as lemmas, the TCCs associated with the definition of `assignS`. But there is a problem: one wants two instantiations of the latter, one for each of the two states at hand. The PVS heuristics for instantiation fare poorly here and must be guided by manually deleting the distracting formulas in the sequent. Reference to these formulas is by line number and this makes the proof script brittle in the face of changes to the prover or the proof.

The noninterference lemma for `mcallS` is the most complex in the entire development. The proof itself is easy, but only because it is factored into three very complicated technical lemmas with long proofs. The semantics of method call is inherently complicated, so these lemmas involves two heaps, two target objects, two argument stores, etc. Each application of the semantics, `mcallS`, generates about eight TCCs as does each use of a lemma like monotonicity of `indis`. The strategy of trying each `mcallS` TCC until one works fails hopelessly here. I have to designate a specific one to use and also the instantiation of its top level variables. But this is not enough. There are nested quantifiers and in the worst case the TCC appears in a sequent with about 35 formulas. An explicit `hide` command is needed to remove half of these so the strategy `smash` can finish the job with simplification and heuristic instantiation. A specialized strategy to automate the `hide` step might work by removing formulas that mention identifiers in which `?` occurs, or those for which there is a matching formula with `?`. But this is neither principled nor straightforward to implement; is there another way?

The noninterference property for commands is as follows.

```
nonint_com: LEMMA
 FORALL L,V,S,(lev:Levels(V)),(me | nonintMenv(me) & writeConfMenv(me)):
     up?( comSafe(V, lev, S) ) => nonint(L, V, lev)( comSem(V,me)(S) )
```

The proof is by structural induction on `S`. For each command construct, the safety condition is unfolded and the noninterference lemma for that construct's semantics is invoked. For if/else, an instantiation is explicitly provided for a lemma on monotonicity of write confinement. But in the rest of the proof the heuristic instantiations work fine and there are no difficulties with TCCs. Finally, the main result is succinctly stated:

```
nonint_CT: THEOREM  safe_CT => nonintMenv( semCT )
```

Like the write confinement lemma for semantics of the class table, the proof is by measure induction (on the approximation chain and on inheritance), using `nonint_com` for method bodies in the induction step. User interaction is needed to introduce lemmas and in a couple of cases provide instantiations.

5 Discussion

The first objective of the project was met: the results of [2] (omitting code-based access control) were checked successfully. The complete import chain for theory `safeProps` comprises about 3K lines of PVS specifications. There are a total of 587 proofs which take approximately 30 minutes to check on a 1.2Ghz Intel CPU with .5G RAM (and took about 8 person-weeks to create). A rough guess for the user written proof scripts is 8K lines.[4]

As for the second objective, the expressiveness of the PVS language certainly lends itself to a semantic model of this sort. The facilities of PVS served well but the `JUDGEMENT` mechanism was of little use and heuristic instantiation of quantifiers leaves something to be desired. But the biggest problem with instantiation is in the contexts where there are two copies of everything in sight, which is likely to pose a problem for any prover and any style of modeling for noninterference.

The effort uncovered a bug in [2]: We had neglected to impose on the semantic domains that the domain of the heap never shrinks, but this was assumed implicitly in some proofs. To my suprise this was the only bug. This is not to say no mistakes were made during the PVS development: In the generalization to an arbitrary lattice of levels I formulated revised definitions off the top of my head, without proving correctness on paper. The third project objective was to find out whether PVS could serve as a proof assistant and help find the correct formulations. I was not disappointed.

The plan is to complete the project by adding access control to the language; this should offer some experience in "proof maintenance". It should be straightforward to extract executable code for the analyzer; this may be used to verify policies created by an inference tool being developed by Qi Sun [17], but it will require extension of the policy language to include level polymorphism. Another interesting project would be to add generics and prove type soundness; on paper the semantics has already been extended [3] to generics as in C# [10]. In joint work with Gary Leavens on foundations of JML [11], we plan to check rules for behavioral subclassing.

Acknowledgements. Natarajan Shankar and Patrick Lincoln arranged a Visiting Fellowship at SRI International for the month of September, 2003, during which time I learned PVS and more than half of this work was carried out. Harold Rueß's elegant domain theoretic work [4] was the leading example for my initial study. Shankar, Sam Owre, and Bruno Dutertre were particularly generous with explanations and advice; Bruno taught me that the way to smash a sequent is to `(smash)`. Walkthroughs by the

[4] A couple of the theories with complicated dependent types run afoul of PVS bugs which hinder use of some tools for exploring proofs. But I have no reason to doubt soundness of the proofs.

whole group, with John Rushby guiding me in fluent emacs-keystroke, were very helpful and encouraging. César Muñoz helped with strategies for TCCs. Thanks to Anindya Banerjee for comments on this paper, not to mention correctness of [2].

References

1. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *15th IEEE Computer Security Foundations Workshop*, 2002.
2. A. Banerjee and D. A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2), 2003. Special issue on Language Based Security.
3. A. Banerjee and D. A. Naumann. State based encapsulation and generics. Technical Report CS-2004-11, Stevens Institute of Technology, 2004.
4. F. Bartels, H. Pfeifer, F. von Henke, and H. Rueß. Mechanizing domain theory. Technical Report UIB-96-10, 1996.
5. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Workshop on Issues in the Theory of Security (WITS)*. ACM, 2003.
6. L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.
7. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23(3):396–459, May 2001.
8. B. Jacobs, W. Pieters, and M. Warnier. Statically checking confidentiality via dynamic labels. In *Workshop on Issues in the Theory of Security (WITS)*. ACM, 2005.
9. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
10. A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation*, 2001.
11. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In *Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852, 2003.
12. P. Levy. Possible world semantics for general storage in call-by-value. In *Computer Science Logic*, LNCS 2471, 2002.
13. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, Saratoga, NY, June 1992.
14. J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI, Dec. 1992.
15. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
16. M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.
17. Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Static Analysis Symposium*, LNCS 3148, 2004.
18. D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, LNCS 1214, 1997.

Proving Bounds for Real Linear Programs in Isabelle/HOL

Steven Obua*

Technische Universität München, D-85748 Garching,
Boltzmannstr. 3, Germany
obua@in.tum.de
http://www4.in.tum.de/~obua

Abstract. Linear programming is a basic mathematical technique for optimizing a linear function on a domain that is constrained by linear inequalities. We restrict ourselves to linear programs on bounded domains that involve only real variables. In the context of theorem proving, this restriction makes it possible for any given linear program to obtain certificates from external linear programming tools that help to prove arbitrarily precise bounds for the given linear program. To this end, an explicit formalization of matrices in Isabelle/HOL is presented, and how the concept of lattice-ordered rings allows for a smooth integration of matrices with the axiomatic type classes of Isabelle.

As our work is a contribution to the Flyspeck project, we argue that with the above techniques it is now possible to prove bounds for the linear programs arising in the proof of the Kepler conjecture sufficiently fast.

1 Introduction and Motivation

The *Flyspeck project* [3] has as its goal the complete formalization of Hales' proof [2] of the Kepler conjecture. The formalization has to be carried out within a mechanical theorem prover. For our work described in this paper, we have chosen the generic proof assistant Isabelle, tailored to Higher-Order Logic (HOL) [4]. In the following, we will refer to this environment as *Isabelle/HOL*.

An important step in Hales' proof is the maximization of about 10^5 *real linear programs*. The size of these linear programs (LPs) varies; the largest among them consist of about 2000 inequalities in about 200 variables. The considered LPs have the important property that there exist a priori bounds on the range of the variables. The situation is further simplified by our attitude towards the linear programs: we only want to know whether the objective function of a given LP is bounded from above by a given constant K.

Under these assumptions, Hales describes [1] a method for obtaining an arbitrarily precise upper bound for the maximum value of the objective function of

* Supported by the Ph.D. program "Logik in der Informatik" of the "Deutsche Forschungsgemeinschaft."

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 227–244, 2005.

an LP. We will show that this method still works nicely in the context of mechanical theorem provers. The burden of calculating the upper bound is delegated to an LP solver that needs not to be trusted. Instead, the LP solver delivers a small certificate to Isabelle/HOL that can be checked cheaply. Furthermore, there is no need to delve into the details of the actual method of optimizing an LP, which is usually the Simplex method. These details just do not matter for the theorem prover.

In this paper we describe all relevant issues and notions that arise in implementing the method in Isabelle/HOL. Although our point of view is necessarily influenced by the capabilities and restrictions of Isabelle/HOL, we think that the results are of independent interest, and we try to present them that way.

We first describe the basic idea of the method. Then we define the notion of *finite matrices* and explain why these are our representation of choice for linear programs. Finite matrices can be fitted into the system of numeric axiomatic type classes in Isabelle/HOL via the algebraic concept of *lattice-ordered rings*, and we take a short look at the changes of the hierarchy of type classes in Isabelle/HOL that were necessary for this. Checking the certificate from the external LP solver is basically a calculation involving finite matrices, and the matrices we have to deal with coming from our Flyspeck background are sparse, therefore we present a sparse matrix representation of finite matrices and formalize operations like sparse matrix multiplication.

2 The Basic Idea

There are quite a lot of different ways to state a linear programming problem [5, sect. 7.4], which are all general in the sense that *every* linear programming problem can be stated that way. Here is one such way: a linear program consists of a matrix $A \in \mathbb{R}^{m\times n}$, a row vector $c \in \mathbb{R}^{1\times n}$ and a column vector $b \in \mathbb{R}^{m\times 1}$. The goal is to maximize the objective function

$$x \longmapsto c\,x, \qquad x \text{ feasible}, \tag{1}$$

where x is called *feasible* iff $x \in \mathbb{R}^{n\times 1}$ and $A\,x \leq b$ holds. Note that we are dealing with matrix inequality here: $X \leq Y$ for two matrices X and Y iff every matrix element of X is less than or equal to the corresponding element of Y.

Usually, the above stated goal really encompasses several goals / questions:

1. Find out if there exists any feasible x at all (otherwise the LP is called *infeasible*).
2. Find out if there is a feasible $x_{\max}$ such that $c\,x_{\max} \geq c\,x$ for any feasible x, and calculate this $x_{\max}$.
3. Calculate $M = \sup\{c\,x \mid x \text{ is feasible}\}$.

Note that $M = -\infty$ iff the answer to the first question is no. And $M = \infty$ iff the answer to the first question is yes and the answer to the second question is no. If $M < \infty$ then the LP is called *bounded*. Linear programming software is good at

answering all those questions and at exhibiting (approximately) such an $x_{\max}$ if it exists. Our goal is more modest in some ways, but more demanding in others: *assuming a priori bounds for the feasible region, that is assuming $l \leq x \leq u$ for all feasible x with a priori known bounds l and u, actually prove within Isabelle/HOL that $M \leq K$, where we can choose K arbitrarily close to M.* In particular, we do not want to calculate $x_{\max}$, but just want to approximate M as precise as we wish for. Furthermore, we can assume $M \neq \infty$ because of

$$M \leq \sum_{i=1}^{n} |c_{1i}| \max\{|l_{i1}|, |u_{i1}|\} < \infty \ . \tag{2}$$

It might seem that our goal can be accomplished trivially by setting K to the above sum. But of course this is not the case, as K is probably not a particularly good approximation for M, and there is nothing in the above inequality telling us how to get a better approximation in case we need one.

2.1 Reducing the Case $M = -\infty$ to the Case $-\infty < M < \infty$

The case of an infeasible LP can be reduced to the case of a feasible LP [1]. We will give a more detailed description here than the one found in [1].

Remember that we are only considering LPs for which we know l and u s.t.

$$A\,x \leq b \Longrightarrow l \leq x \leq u \ . \tag{3}$$

In this subsection we additionally require A to fulfill the inequality

$$A\,x \leq 0 \Longrightarrow x = 0 \ . \tag{4}$$

This can easily be arranged by replacing A and b by $\tilde{A}$ and $\tilde{b}$ where

$$\tilde{A} = \begin{pmatrix} A \\ I_n \\ -I_n \end{pmatrix} \quad \text{and} \quad \tilde{b} = \begin{pmatrix} b \\ u \\ -l \end{pmatrix} \ . \tag{5}$$

$I_n \in \mathbb{R}^{n \times n}$ denotes the identity matrix.

Now let us assume that for the given LP both (3) and (4) hold. We can construct for any $K \in \mathbb{R}$ a modified LP with objective function

$$x' = \begin{pmatrix} x \\ t \end{pmatrix} \longmapsto c\,x + K\,t, \quad x' \text{ feasible}, \tag{6}$$

where $x' \in \mathbb{R}^{n+1}$ is called feasible with respect to the modified LP iff

$$A\,x + t\,b \leq b \quad \text{and} \quad 0 \leq t \leq 1 \ . \tag{7}$$

Lemma 1.

$$\binom{x}{1} \text{ is feasible} \iff x = 0 \ , \tag{8}$$

$$0 \leq t < 1 \implies \left(\binom{x}{t} \text{ is feasible} \iff x/(1-t) \text{ is feasible} \right) \ . \tag{9}$$

On the left hand side of above equivalences we talk about feasibility with respect to the modified LP, on the right hand side about feasibility with respect to the original LP.

Proof. To show (8) in the direction from left to right one needs the fact that A fulfills (4). The rest ist obvious by just expanding the respective definition of feasibility. □

Lemma 2. *Defining* $M' := \sup\{c\,x + K\,t \mid x' = \binom{x}{t},\ x' \text{ feasible}\}$ *yields*

$$-\infty < \max\{M, K\} = M' < \infty \ . \tag{10}$$

As a special case follows

$$M = -\infty \implies M' = K \ . \tag{11}$$

Proof. Because of (8) we have $M' \geq K$, in particular $M' > -\infty$. Considering $t = 0$ in (9) gives us $M' \geq M$. From (9) and (3) in the case $t \neq 1$ and (8) in the case $t = 1$ we obtain bounds for x':

$$x' = \binom{x}{t} \text{ is feasible } \implies l^- \leq (1-t)\,l \leq x \leq (1-t)\,u \leq u^+ \ .$$

Here l^- denotes the *negative part* of l which results from l by replacing every positive matrix element by 0. Similarly, the *positive part* u^+ results from u by replacing every negative element by 0. We conclude $M' < \infty$.

So far we have shown $-\infty < \max\{M, K\} \leq M' < \infty$. To complete the proof, we need to show $\max\{M, K\} \geq M'$. We will proceed by case distinction.

Assume $M \geq K$. We show that for any feasible $x' = \binom{x}{t}$, $M \geq c\,x + K\,t$, and therefore $M \geq M'$. This is obvious in the case $t = 1$, the feasibility of x' accompanied by the equivalence (8) forces x to be zero. In the case $t \neq 1$, (9) implies that $x/(1-t)$ is feasible with respect to the original LP. But this is just what we claim:

$$M \geq c\,(x/(1-t)) \implies M \geq c\,x + t\,M \geq c\,x + t\,K \ .$$

Now assume $M < K$. Assume further $M' > K$. Because of $-\infty < M' < \infty$ there is a feasible $x' = \binom{x}{t}$ s.t. $M' = c\,x + K\,t$. For $t = 1$ we would have again $x = 0$ and therefore the contradiction $K < M' = K$. Finally $0 \leq t < 1$ also leads to a contradiction:

$$M' = c\,x + K\,t \leq c\,x + M'\,t \implies M' \leq c\,(x/(1-t)) \leq M < K \leq M' \ .$$

Therefore the only possibility is $M' \leq K$. □

From now on we will assume that we are dealing with feasible, bounded LPs, that is with LPs for which we know $-\infty < M < \infty$.

2.2 The Case $-\infty < M < \infty$

This case is the heart of the method. Again we construct a modified LP. The original LP is called the *primal* LP, the modified LP is called the *dual* LP. The objective function of the dual LP

$$y \longmapsto y\,b, \qquad y \text{ feasible},$$

is to be minimized. Here $y \in \mathbb{R}^{1\times m}$ is called feasible iff $yA = c$ and $y \geq 0$ holds.

Lemma 3. *Any feasible y induces an upper bound on M:*

$$y\,b \;\geq\; M \;. \tag{12}$$

Proof. For any feasible x we have

$$y\,b \geq y\,(A\,x) = (y\,A)\,x = c\,x \;. \tag{13}$$

□

But is there such a feasible y so that we can utilize (12)? And if there is, can we accomplish $y\,b = M$ by carefully choosing y? The well-known answer to both questions is yes:

Lemma 4. *Define $M' := \inf\{y\,b \,|\, y\,A = c \text{ and } y \geq 0\}$. Then*

$$-\infty \;<\; M = M' \;<\; \infty \;. \tag{14}$$

Furthermore, choose a feasible y such that $M' = y\,b$. Then

$$\operatorname{card}\{i \in \mathbb{N} \,|\, 1 \leq i \leq m \text{ and } y_{1i} \;>\; 0\} \;\leq\; n \;. \tag{15}$$

Proof. Corollary 7.1g and 7.1l in [5]. □

Now the basic idea of our method can be described as follows. First, form the dual LP. Then use an external LP solver to solve the dual LP for an optimal y. This optimal y serves as a *certificate*. In our application, where typically $m \approx 2000$ and $n \approx 200$, y will be *sparse*, as inequality (15) tells us. Finally, use (12) to verify our desired upper bound $M \leq K = y\,b$.

This basic idea is complicated by the fact that we are dealing with *real* data and *numerical* algorithms. The external LP solver does not return an y such that $y\,A = c$ and $y \geq 0$, but rather an y such that $y\,A \approx c$ and $y \gtrapprox 0$. In order to obtain a provably upper bound on M, one has to take (3) into account. Furthermore the input data A, b and c need not to be given as exact numerical data either, for example an element of A could equal π.

The rest of the paper will discuss the implementation in Isabelle/HOL of the method outlined here and will also deal with the mentioned complications.

3 Finite Matrices

Somebody who wants to implement the method outlined in the previous section faces up to the problem of how to represent linear programs. This problem is prominent outside of the realm of mechanical theorem proving, too: designers of linear programming packages typically provide various ways of input of data to the LP algorithms these packages provide, one can normally choose at least between dense and sparse representations of the data. The issue is to provide a certain convenience of dealing with the data without compromising the efficiency of the LP algorithms by too much overhead.

Our situation is different: we need to reason within our mechanical theorem proving environment Isabelle/HOL why our computations lead to a correct result, therefore we need a good representation of LPs for reasoning about them. Of course we also need to compute efficiently. But we should avoid mixing up those two issues if we can. The reasoning in the previous section has used matrices and the properties of matrix operations like associativity of matrix multiplication extensively. Therefore representing LPs within Isabelle/HOL as matrices is a good idea.

So how exactly does one represent matrices in higher-order logic? Obviously, matrices should be a type, but how does one deal with the dimension of a matrix? HOL does not have *dependent types*, so it seems impossible to have a parametrized family of types where the dimension of the matrix would be the parameter. But it is: one possibility that is pursued by John Harrison in the 2005 version of his Hol-light system is to represent the needed parameter by type variables! He uses this representation in order to formalize multivariate calculus. But in our case this idea cannot be used without causing serious problems later when we turn our attention to sparse matrices.

Another possibility is to represent the dimension of a matrix by a predicate that carves the set of all matrices of this dimension out of a certain bigger, already existing type. This is a common technique to overcome the absence of dependent types in HOL [10]. This approach could work like this:[1]

$$\begin{array}{l}
\textbf{type}\ \alpha\ M = nat \times nat \times (nat \Rightarrow nat \Rightarrow \alpha) \\
\\
\textbf{constdef} \\
\quad Mequiv :: (\alpha\ M * \alpha\ M)\ set \\
\quad Mequiv \equiv \{((m,n,f),(m,n,g)) \,|\, \forall\, j\, i.\, (j < m \wedge i < n) \longrightarrow f\, j\, i = g\, j\, i\} \\
\\
\textbf{typedef}\ \alpha\ matrix = UNIV\ //\ Mequiv \\
\\
\textbf{constdef} \\
\quad is\text{-}matrix :: nat \Rightarrow nat \Rightarrow \alpha\ matrix \Rightarrow bool \\
\quad is\text{-}matrix\ m\ n\ A \equiv \exists\, f.\, (m,n,f) \in Rep\text{-}matrix\, A.
\end{array} \tag{16}$$

[1] Here and in the following we deviate slightly from actual Isabelle/HOL syntax for various reasons, the most important being formatting; the actual Isabelle/HOL user will have no difficulty translating the given theory snippets to proper Isabelle/HOL syntax.

In (16) α *matrix* is the bigger type, and *is-matrix* m n acts as the predicate that carves out all matrices consisting of m rows and n columns. Here matrices are modelled as equivalence classes [7] of triples (m, n, f) where m denotes the number of rows, n the number of columns and f a function from indices to matrix elements. The set of these equivalence classes is denoted by *UNIV // Mequiv*. With this formalization of matrices an error element comes for free: there is exactly one matrix *Error* such that

$$\textit{is-matrix}\,0\,0\,\textit{Error} \tag{17}$$

holds. When adding matrices A and B which fulfill

$$\exists\ m\ n\ .\ (\textit{is-matrix}\ m\ n\ A) \wedge (\neg\ \textit{is-matrix}\ m\ n\ B) \tag{18}$$

and when multiplying matrices A and B for which

$$\exists\ m\ n\ u\ v.\ (\textit{is-matrix}\ m\ n\ A) \wedge (\textit{is-matrix}\ u\ v\ B) \wedge (n \neq u) \tag{19}$$

holds, the matrix *Error* is returned to signal that the operands do not belong to the natural domain of addition and multiplication, respectively.

Still, this approach is not entirely satisfying: in Isabelle/HOL there exists a large number of theorems that are valid for types that form a group or a ring. The fact that a type can be viewed as such an algebraic structure is formulated via the concept of *axiomatic type classes* [6]. But α *matrix* in (16) with the suggested error signaling definition of addition does not even form a group, because there is no matrix *Zero* with

$$\forall\, A.\, A + \textit{Zero} = A\ , \tag{20}$$

but rather a whole family $\textit{Zero}\,m\,n$ such that

$$\forall\, A.\, \textit{is-matrix}\, m\, n\, A \longrightarrow A + (\textit{Zero}\, m\, n) = A\ . \tag{21}$$

Therefore we advocate a different approach that exploits the fact that the matrix elements commonly used in mathematics [11] themselves carry an algebraic structure, that of a ring, which always contains a zero. We define α *matrix* to be the type formed by all infinite matrices that have only finitely many non-zero elements of type α:

$$\begin{aligned}&\textbf{type}\ \alpha\ \textit{infmatrix} = \textit{nat} \Rightarrow \textit{nat} \Rightarrow \alpha\\ &\textbf{typedef}\ \alpha\ \textit{matrix} = \{f :: (\alpha :: \textit{zero})\textit{infmatrix} \mid \textit{finite}\,\{(j,i) \mid f\ j\ i \neq 0\}\}\ .\end{aligned} \tag{22}$$

Hence we choose the name *finite matrices* for objects of type α *matrix*. Note the restriction $\alpha :: \textit{zero}$ in (22). This means that the elements of a matrix cannot have just any type but only a type that is an instance of the axiomatic type class *zero* and has thus an element denoted by 0. Of course this is not a real restriction on the type; any type can be declared to be an instance of the axiomatic type class *zero*.

3.1 Dimension of a Finite Matrix

The dimension of a finite matrix deviates from the notion of dimension that one is used to. Because we did *not* encode the number of rows and columns explicitly in the representation of a finite matrix as we did in (16), we have to recover the dimension of a finite matrix by extensionality:

$$\begin{array}{l}\textbf{constdefs}\\ \quad \mathit{nrows} :: \alpha\ \mathit{matrix} \Rightarrow \mathit{nat}\\ \quad \mathit{nrows}\ A \equiv \mathit{LEAST}\, m.\, \forall\, j\, i.\, m \leq j \longrightarrow (\mathit{Rep\text{-}matrix}\, A\, j\, i = 0)\\ \quad \mathit{ncols} :: \alpha\ \mathit{matrix} \Rightarrow \mathit{nat}\\ \quad \mathit{ncols}\ A \equiv \mathit{LEAST}\, n.\, \forall\, j\, i.\, n \leq i \longrightarrow (\mathit{Rep\text{-}matrix}\, A\, j\, i = 0)\\ \quad \mathit{is\text{-}matrix} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \alpha\ \mathit{matrix} \Rightarrow \mathit{bool}\\ \quad \mathit{is\text{-}matrix}\ m\ n\ A \equiv \mathit{nrows}\, A \leq m \wedge \mathit{ncols}\, A \leq n\ .\end{array} \tag{23}$$

The expression $\mathit{LEAST}\, x.\, P\, x$ equals the least x such that $P\, x$ holds. The definition of the type $\alpha\ \mathit{matrix}$ has introduced two automatically defined functions *Rep-matrix* and *Abs-matrix*

$$\begin{array}{l}\textbf{consts}\\ \quad \mathit{Rep\text{-}matrix} :: \alpha\ \mathit{matrix} \Rightarrow \alpha\ \mathit{infmatrix}\\ \quad \mathit{Abs\text{-}matrix} :: \alpha\ \mathit{infmatrix} \Rightarrow \alpha\ \mathit{matrix}\end{array} \tag{24}$$

that convert between finite matrices and infinite matrices. They enjoy the following crucial properties:

$$(A = B) = (\forall\, j\, i.\, \mathit{Rep\text{-}matrix}\, A\, j\, i = \mathit{Rep\text{-}matrix}\, B\, j\, i)\ , \tag{25}$$

$$\exists_1\, f.\, A = \mathit{Abs\text{-}matrix}\, f\ , \tag{26}$$

$$\mathit{Abs\text{-}matrix}\, (\mathit{Rep\text{-}matrix}\, A) = A\ , \tag{27}$$

$$\mathit{finite}\, \{(j, i) \mid \mathit{Rep\text{-}matrix}\, A\, j\, i \neq 0\}\ , \tag{28}$$

$$\mathit{finite}\, \{(j, i) \mid f\, j\, i \neq 0\} \Longrightarrow \mathit{Rep\text{-}matrix}\, (\mathit{Abs\text{-}matrix}\, f) = f\ . \tag{29}$$

Thus $\mathit{Rep\text{-}matrix}\, A\, j\, i$ denotes the matrix element of A in row j and column i. Note that the first row is row 0, likewise for columns.

Let us return to the definitions in (23). The definition of *is-matrix* implies that a matrix has not exactly one dimension, but infinitely many! Therefore there is no need for signaling an error due to incompatibility of dimensions: for any two matrices A and B one shows

$$\exists\, m.\, \mathit{is\text{-}matrix}\, m\, m\, A \wedge \mathit{is\text{-}matrix}\, m\, m\, B\ . \tag{30}$$

The intuition behind (30) is that every matrix can be viewed as a square matrix of dimension m as long as m is large enough: one just needs to fill up the missing rows and columns with zeros.

The need for an *Error* matrix has vanished, but one can still use (17) to uniquely define a matrix. This time, we denote that matrix by 0:

$$\forall A.\,(A = 0) = (\textit{is-matrix}\,0\,0\,A)\ . \tag{31}$$

Another possibility of defining 0 is given by the following theorem:

$$\forall A.\,(A = 0) = (\forall m\,n.\,\textit{is-matrix}\,m\,n\,A)\ . \tag{32}$$

We will see that 0 is actually the proper name for this matrix.

3.2 Lifting Unary Operators

In this subsection we look at how to define an unary operator U on matrices,

$$U :: \alpha\ \textit{matrix} \Rightarrow \beta\ \textit{matrix}\ , \tag{33}$$

by lifting an unary operator u on matrix elements,

$$u :: \alpha \Rightarrow \beta\ . \tag{34}$$

The first step is to lift u to infinite matrices:

constdef
$$\begin{aligned}&\textit{apply-infmatrix} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha\ \textit{infmatrix} \Rightarrow \beta\ \textit{infmatrix})\\ &\textit{apply-infmatrix}\ u \equiv \lambda f\,j\,i.\,u\,(f\,j\,i)\ ,\end{aligned} \tag{35}$$

which results in the lifting property

$$(\textit{apply-infmatrix}\,u\,f)\,j\,i = u\,(f\,j\,i)\ . \tag{36}$$

Its proof is apparent from the definition of *apply-infmatrix*.

Now the unary lifting operator *apply-matrix* can be defined by first lifting u to infinite matrices, and then to finite matrices:

constdef
$$\begin{aligned}&\textit{apply-matrix} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha\ \textit{matrix} \Rightarrow \beta\ \textit{matrix})\\ &\textit{apply-matrix}\ u \equiv \lambda A.\,\textit{Abs-matrix}\,(\textit{apply-infmatrix}\,u\,(\textit{Rep-matrix}\ A))\ .\end{aligned} \tag{37}$$

What is the lifting property for *apply-matrix*? A first guess yields

$$\textit{Rep-matrix}\,(\textit{apply-matrix}\,u\,A)\,j\,i = u\,(\textit{Rep-matrix}\,A\,j\,i)\ . \tag{38}$$

But this is false (in the sense that we cannot prove it in HOL)! To see why, consider $\alpha = \beta = \textit{int}$ and $u = \lambda x.\,1$. Then we have

$$\textit{apply-infmatrix}\,u\,(\textit{Rep-matrix}\,A) = \begin{pmatrix}1 & 1 & \cdots\\ 1 & 1 & \cdots\\ \vdots & \vdots & \ddots\end{pmatrix} \neq \textit{Rep-matrix}\,B \tag{39}$$

for all matrices A and any matrix B. But there is a simple condition on u that turns out to be sufficient and necessary to prove (38):

$$u\,0 = 0 \Longrightarrow \textit{Rep-matrix}\,(\textit{apply-matrix}\,u\,A)\,j\,i = u\,(\textit{Rep-matrix}\,A\,j\,i)\ . \tag{40}$$

This is easily provable using (37), (28), (29) and (36).

3.3 Lifting Binary Operators

Just as we have defined a unary lifting operator *apply-matrix*, we can define similarly a binary lifting operator *combine-matrix*:

$$\begin{array}{l}\textbf{constdef}\\ \quad \mathit{combine\text{-}infmatrix} :: \\ \qquad (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\alpha\ \mathit{infmatrix} \Rightarrow \beta\ \mathit{infmatrix} \Rightarrow \gamma\ \mathit{infmatrix})\\ \quad \mathit{combine\text{-}infmatrix}\ v \equiv \lambda f\, g\, j\, i.\, v\,(f\, j\, i)\,(g\, j\, i)\ ,\end{array} \tag{41}$$

$$\begin{array}{l}\textbf{constdef}\\ \quad \mathit{combine\text{-}matrix} :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\alpha\ \mathit{matrix} \Rightarrow \beta\ \mathit{matrix} \Rightarrow \gamma\ \mathit{matrix})\\ \quad \mathit{combine\text{-}matrix}\ v \equiv \\ \qquad \lambda A\, B.\, \mathit{Abs\text{-}matrix}\,(\mathit{combine\text{-}infmatrix}\, v\,(\mathit{Rep\text{-}matrix}\, A)\,(\mathit{Rep\text{-}matrix}\, B))\ ,\end{array} \tag{42}$$

The lifting property for *combine-matrix* reads

$$\begin{array}{l} v\,0\,0 = 0 \Longrightarrow \mathit{Rep\text{-}matrix}\,(\mathit{combine\text{-}matrix}\, v\, A\, B)\, j\, i = \\ \qquad\qquad v\,(\mathit{Rep\text{-}matrix}\, A\, j\, i)\,(\mathit{Rep\text{-}matrix}\, B\, j\, i)\ .\end{array} \tag{43}$$

Lifting binary operators passes on commutativity and associativity. Defining

$$\begin{array}{l}\textbf{constdefs}\\ \quad \mathit{commutative} :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \mathit{bool}\\ \quad \mathit{commutative}\ v \equiv \forall x\, y.\, v\, x\, y = v\, y\, x\\ \quad \mathit{associative} :: (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \mathit{bool}\\ \quad \mathit{associative}\ v \equiv \forall x\, y\, z.\, v\,(v\, x\, y)\, z = v\, x\,(v\, y\, z)\end{array} \tag{44}$$

we can formulate this propagation concisely:

$$\begin{array}{l} \mathit{commutative}\, v \Longrightarrow \mathit{commutative}\,(\mathit{combine\text{-}matrix}\, v)\ ,\\ [\![v\,0\,0 = 0;\ \mathit{associative}\, v]\!] \Longrightarrow \mathit{associative}\,(\mathit{combine\text{-}matrix}\, v)\ .\end{array} \tag{45}$$

You might be surprised that the propagation of commutativity does not require $v\,0\,0 = 0$, which is due to the idiosyncrasies of the definite description operator that is hidden in *Abs-matrix*.

3.4 Matrix Multiplication

We need one last lifting operation, the most interesting one: given two binary operators addition and multiplication on the matrix elements, define the matrix product induced by those two operators. As a basic tool we first define by primitive recursion a fold operator that acts on sequences:

$$\begin{array}{l}\textbf{const}\ \mathit{foldseq} :: (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow (\mathit{nat} \Rightarrow \alpha) \Rightarrow \mathit{nat} \Rightarrow \alpha\\ \textbf{primrec}\\ \quad \mathit{foldseq}\ f\ s\ 0 = s\,0\\ \quad \mathit{foldseq}\ f\ s\ (\mathit{Suc}\, n) = f\,(s\,0)\,(\mathit{foldseq}\, f\,(\lambda k.\, s\,(\mathit{Suc}\, k))\, n)\end{array} \tag{46}$$

For illustration purposes, assume $s = (s_1, s_2, s_3, s_4, \ldots, s_n, 0, 0, 0, \ldots)$. Then

$$\begin{array}{l} \mathit{foldseq}\ f\ s\ 0 = s_1\ ,\\ \mathit{foldseq}\ f\ s\ 1 = f\, s_1\, s_2\ ,\\ \mathit{foldseq}\ f\ s\ 2 = f\, s_1\,(f\, s_2\, s_3)\ ,\\ \mathit{foldseq}\ f\ s\ 3 = f\, s_1\,(f\, s_2\,(f\, s_3\, s_4))\ ,\\ \mathit{foldseq}\ f\ s\ n = f\, s_1\,(f\, s_2\,(\ldots\,(f\, s_n\, 0)\ldots))\ ,\\ \mathit{foldseq}\ f\ s\ (n+1) = f\, s_1\,(f\, s_2\,(\ldots\,(f\, s_n\,(f\,0\,0)\ldots))\ \text{and so on.}\end{array} \tag{47}$$

Note that if $f\,0\,0 = 0$ the above sequence converges:

$$f\,0\,0 = 0 \Longrightarrow \forall\, m.\, n \leq m \longrightarrow \mathit{foldseq}\, f\, s\, m = \mathit{foldseq}\, f\, s\, n \ . \tag{48}$$

Now we are prepared to deal with matrix multiplication:

constdef
$$\begin{aligned}&\mathit{mult\text{-}matrix\text{-}n} :: \mathit{nat} \Rightarrow (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\gamma \Rightarrow \gamma \Rightarrow \gamma) \Rightarrow \\ &\qquad \alpha\ \mathit{matrix} \Rightarrow \beta\ \mathit{matrix} \Rightarrow \gamma\ \mathit{matrix} \\ &\mathit{mult\text{-}matrix\text{-}n}\ n\ \mathit{mult}\ \mathit{add}\ A\ B \equiv \mathit{Abs\text{-}matrix}\ (\lambda\, j\, i. \\ &\quad \mathit{foldseq}\, \mathit{add}\, (\lambda\, k.\, \mathit{mult}\, (\mathit{Rep\text{-}matrix}\, A\, j\, k)\, (\mathit{Rep\text{-}matrix}\, B\, k\, i))\, n)\end{aligned} \tag{49}$$

The idea of *mult-matrix-n n mult add A B* is to consider only the first n columns of A and the first n rows of B when calculating the matrix product. Of course the matrix product should be independent of n. We achieve this by setting

$$\mathit{mult\text{-}matrix}\, \mathit{mult}\, \mathit{add} \equiv \lim_{n \to \infty}\ \mathit{mult\text{-}matrix\text{-}n}\, n\, \mathit{mult}\, \mathit{add}\ , \tag{50}$$

which is due to (48) well-defined if $\forall\, x.\, \mathit{mult}\, x\, 0 = \mathit{mult}\, 0\, x = \mathit{add}\, 0\, 0 = 0$ holds:

constdef
$$\begin{aligned}&\mathit{mult\text{-}matrix} :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\gamma \Rightarrow \gamma \Rightarrow \gamma) \Rightarrow \\ &\qquad \alpha\ \mathit{matrix} \Rightarrow \beta\ \mathit{matrix} \Rightarrow \gamma\ \mathit{matrix} \\ &\mathit{mult\text{-}matrix}\ \mathit{mult}\ \mathit{add}\ A\ B \equiv \\ &\quad \mathit{mult\text{-}matrix\text{-}n}\, (\max\, (\mathit{ncols}\, A)\, (\mathit{nrows}\, B))\, \mathit{mult}\, \mathit{add}\, A\, B\ .\end{aligned} \tag{51}$$

Again, we have a lifting property:

$$\begin{aligned}&[\![\, \forall\, x.\, \mathit{mult}\, x\, 0 = 0 \wedge \mathit{mult}\, 0\, x = 0;\ \mathit{add}\, 0\, 0 = 0\,]\!] \Longrightarrow \\ &\quad \mathit{Rep\text{-}matrix}\, (\mathit{mult\text{-}matrix}\, \mathit{mult}\, \mathit{add}\, A\, B)\, j\, i = \mathit{foldseq}\, \mathit{add} \\ &\qquad (\lambda\, k.\, \mathit{mult}\, (\mathit{Rep\text{-}matrix}\, A\, j\, k)\, (\mathit{Rep\text{-}matrix}\, B\, k\, i))\, (\max\, (\mathit{ncols}\, A)\, (\mathit{nrows}\, B))\ .\end{aligned} \tag{52}$$

Finally, let us examine what properties of element addition and element multiplication induce distributivity and associativity of *mult-matrix*.

Distributivity. We distinguish between left and right distributivity:[2]

constdefs
$$\begin{aligned}&\mathit{r\text{-}distributive} :: (\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \beta \Rightarrow \beta) \Rightarrow \mathit{bool} \\ &\mathit{r\text{-}distributive}\ \mathit{mult}\ \mathit{add} \equiv \forall\, a\, u\, v.\, \mathit{mult}\, a\, (\mathit{add}\, u\, v) = \mathit{add}\, (\mathit{mult}\, a\, u)\, (\mathit{mult}\, a\, v) \\ &\mathit{l\text{-}distributive} :: (\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \mathit{bool} \\ &\mathit{l\text{-}distributive}\ \mathit{mult}\ \mathit{add} \equiv \forall\, a\, u\, v.\, \mathit{mult}\, (\mathit{add}\, u\, v)\, a = \mathit{add}\, (\mathit{mult}\, u\, a)\, (\mathit{mult}\, v\, a)\end{aligned} \tag{53}$$

Distributivity of *mult* over *add* lifts to distributivity of *mult-matrix mult add* over *combine-matrix add* if *add* is associative and commutative and both *add* and *mult* behave as expected with respect to 0:

[2] Our convention is that left distributivity means that the factor is distributed over the left sum, *not* that the left factor is the one that gets distributed.

$$\begin{array}{l}
[\![\, \textit{l-distributive mult add; associative add; commutative add;} \\
\quad \forall x.\, \textit{mult}\, x\, 0 = 0 \wedge \textit{mult}\, 0\, x = 0;\ \textit{add}\, 0\, 0 = 0 \,]\!] \\
\Longrightarrow \textit{l-distributive}\ (\textit{mult-matrix mult add})\ (\textit{combine-matrix add})\ , \\
[\![\, \textit{r-distributive mult add; associative add; commutative add;} \\
\quad \forall x.\, \textit{mult}\, x\, 0 = 0 \wedge \textit{mult}\, 0\, x = 0;\ \textit{add}\, 0\, 0 = 0 \,]\!] \\
\Longrightarrow \textit{r-distributive}\ (\textit{mult-matrix mult add})\ (\textit{combine-matrix add})\ .
\end{array} \tag{54}$$

Associativity. We state the law of associativity for *mult-matrix* in a very general form:

$$\begin{array}{l}
[\![\, \forall a.\, \textit{mult}_1\, a\, 0 = 0;\ \forall a.\, \textit{mult}_1\, 0\, a = 0;\ \forall a.\, \textit{mult}_2\, a\, 0 = 0;\ \forall a.\, \textit{mult}_2\, 0\, a = 0; \\
\quad \textit{add}_1\, 0\, 0 = 0;\ \textit{add}_2\, 0\, 0 = 0; \\
\quad \forall a\, b\, c\, d.\, \textit{add}_2\, (\textit{add}_1\, a\, b)\, (\textit{add}_1\, c\, d) = \textit{add}_1\, (\textit{add}_2\, a\, c)\, (\textit{add}_2\, b\, d); \\
\quad \forall a\, b\, c.\, \textit{mult}_2\, (\textit{mult}_1\, a\, b)\, c = \textit{mult}_1\, a\, (\textit{mult}_2\, b\, c); \\
\quad \textit{associative add}_1; \textit{associative add}_2; \\
\quad \textit{l-distributive mult}_2\ \textit{add}_1; \textit{r-distributive mult}_1\ \textit{add}_2\ \,]\!] \\
\Longrightarrow\ \textit{mult-matrix mult}_2\ \textit{add}_2\ (\textit{mult-matrix mult}_1\ \textit{add}_1\ A\ B)\ C = \\
\qquad \textit{mult-matrix mult}_1\ \textit{add}_1\ A\ (\textit{mult-matrix mult}_2\ \textit{add}_2\ B\ C)\ .
\end{array} \tag{55}$$

For $\textit{mult} = \textit{mult}_1 = \textit{mult}_2$ and $\textit{add} = \textit{add}_1 = \textit{add}_2$ this simplifies to

$$\begin{array}{l}
[\![\, \forall a.\, \textit{mult}\, a\, 0 = 0;\ \forall a.\, \textit{mult}\, 0\, a = 0;\ \textit{add}\, 0\, 0 = 0; \\
\quad \textit{associative add; commutative add; associative mult;} \\
\quad \textit{l-distributive mult add; r-distributive mult add}\ \,]\!] \\
\Longrightarrow\ \textit{associative}\ (\textit{mult-matrix mult add})\ .
\end{array} \tag{56}$$

3.5 Lattice-Ordered Rings

Paulson describes in [6] how numerical theories like the theory of integers or the theory of reals can be organized in Isabelle/HOL using axiomatic type classes. For example both integers and reals form a ring, therefore Paulson recommends to prove theorems that are implied purely by ring properties only once, and then to prove that both types *int* and the type *real* are an instance of the axiomatic type class *ring*.

Birkhoff points out [12, chapt. 17] that for a fixed n the ring of all $n \times n$ square matrices forms a latticed-ordered ring in a natural way. The same is true for our finite matrices! Therefore it suggests itself to establish an axiomatic type class *lordered-ring* that captures the property of a type to form a lattice-ordered ring. Of course *lordered-ring* should be integrated with the other type classes like *ring* and *ordered-ring* of Isabelle/HOL to maximize theorem reuse. Two major changes along with minor modifications were necessary to the original hierarchy of type classes as described in [6]:

1. The original type class *ring* demanded both the existence of a multiplicative unit element and the commutativity of multiplication. But our finite matrices do not have such a multiplicative unit element, nor is multiplication of finite matrices a commutative operator. Nevertheless, finite matrices still form a ring in common mathematical terminology. Therefore the original type class

ring was renamed to become *comm-ring-1* and new type classes *ring*, *ring-1* and *comm-ring* were introduced, suitable for rings that do not necessarily possess a 1 and/or are not commutative.

2. All ordered algebraic structures contained in the original hierarchy were linearly ordered. The natural (elementwise) order for finite matrices is a proper partial order, actually a lattice order. Therefore we enriched the hierarchy with type classes that model partially ordered algebraic systems like partially ordered groups and rings, or lattice-ordered groups and rings. For this we follow largely [13], [12].

We do not want to delve into the details of the modified hierarchy, but refer the curious reader to the 2005 release of Isabelle where all these changes have been incorporated. Instead, let us directly turn to lattice-ordered rings. A type α is an instance of the axiomatic type class *lordered-ring* iff

ring. α is a ring with addition $+$, subtraction $-$, additive inverse $-$, multiplication $*$, zero 0,
lattice. α is a lattice with partial order $\leq$ and operators *join* and *meet*,
monotonicity. addition and multiplication are monotone:

$$a \leq b \longrightarrow c + a \leq c + b \ , \tag{57}$$
$$a \leq b \wedge 0 \leq c \longrightarrow a * c \leq b * c \wedge c * a \leq c * b \ . \tag{58}$$

Both *int* and *real* are instances of *lordered-ring*:

$$\begin{array}{l}\textbf{instance}\ \mathit{int} :: \mathit{lordered\text{-}ring} \\ \textbf{instance}\ \mathit{real} :: \mathit{lordered\text{-}ring}\ .\end{array} \tag{59}$$

Our goal is to prove

$$\textbf{instance}\ \mathit{matrix} :: (\mathit{lordered\text{-}ring})\ \mathit{lordered\text{-}ring}\ . \tag{60}$$

The above meta theorem has the following meaning (which is not legal Isabelle syntax):

$$(\textbf{instance}\ \alpha :: \mathit{lordered\text{-}ring}) \Longrightarrow (\textbf{instance}\ \alpha\ \mathit{matrix} :: \mathit{lordered\text{-}ring})\ . \tag{61}$$

Of course, in order to prove (60), one first has to define 0, $+$, $*$ etc. for objects of type *matrix*. The zero matrix is easy to define:

$$\begin{array}{l}\textbf{instance}\ \mathit{matrix} :: (\mathit{zero})\ \mathit{zero} \\ \textbf{def (overloaded)} \\ \quad 0 \equiv \mathit{Abs\text{-}matrix}\,(\lambda\, j\, i.\, 0)\ .\end{array} \tag{62}$$

It is simple to show that this is actually the 0 we refer to in (31) and (32).

Addition $+$, multiplication $*$, subtraction $-$, unary minus $-$, can all be defined using the lifting machinery we have developed:

$$\begin{array}{l}\textbf{instance}\ \mathit{matrix} :: (\mathit{plus})\ \mathit{plus}\\ \textbf{instance}\ \mathit{matrix} :: (\mathit{minus})\ \mathit{minus}\\ \textbf{instance}\ \mathit{matrix} :: (\{\mathit{plus}, \mathit{times}\})\ \mathit{times}\\ \textbf{defs (overloaded)}\\ \quad A + B \equiv \mathit{combine\text{-}matrix}\,(\lambda\, a\, b.\, a + b)\, A\, B\\ \quad A - B \equiv \mathit{combine\text{-}matrix}\,(\lambda\, a\, b.\, a - b)\, A\, B\\ \quad\ -A \equiv \mathit{apply\text{-}matrix}\,(\lambda\, a.\ -a)\, A\\ \quad A * B \equiv \mathit{mult\text{-}matrix}\,(\lambda\, a\, b.\, a * b)\,(\lambda\, a\, b.\, a + b)\, A\, B\ .\end{array} \tag{63}$$

Finally, we need to be able to compare matrices:

$$\begin{array}{l}\textbf{instance}\ \mathit{matrix} :: (\{\mathit{ord}, \mathit{zero}\})\ \mathit{ord}\\ \textbf{defs (overloaded)}\\ \quad A \leq B \equiv \forall\, j\, i.\, \mathit{Rep\text{-}matrix}\, A\, j\, i \leq \mathit{Rep\text{-}matrix}\, B\, j\, i\end{array} \tag{64}$$

After having introduced the necessary syntax, we need to show that $\alpha\ \mathit{matrix}$ really constitutes a lattice-ordered ring, provided α constitutes one, in order to obtain (60). But almost the entire work has already been done: for example, in order to prove associativity of matrix multiplication,

$$\forall\,(A :: (\alpha :: \mathit{lordered\text{-}ring})\ \mathit{matrix}).\, A * (B * C) = (A * B) * C\ , \tag{65}$$

which is the hardest of all proof obligations, just apply (56)! The remaining proof obligations are not difficult to prove, either, one just has to make use of matrix extensionality (25) and the lifting properties (40), (43) and (52). It is useful, though, first to dispose of the assumptions in these lifting properties, so for example instead of using (43) directly one should prove and use

$$\begin{array}{l}\mathit{Rep\text{-}matrix}\,(A + B)\, j\, i = (\mathit{Rep\text{-}matrix}\, A\, j\, i) + (\mathit{Rep\text{-}matrix}\, B\, j\, i)\\ \mathit{Rep\text{-}matrix}\,(A - B)\, j\, i = (\mathit{Rep\text{-}matrix}\, A\, j\, i) - (\mathit{Rep\text{-}matrix}\, B\, j\, i)\ .\end{array} \tag{66}$$

A proof obligation that differs from the others because it is not a universal property that needs to be shown, but an existential one, turns up when one has to show that *join* and *meet* do exist:

$$\begin{array}{l}\exists\, j.\, \forall\, a\, b\, x.\, a \leq j\, a\, b \wedge b \leq j\, a\, b \wedge (a \leq x \wedge b \leq x \longrightarrow j\, a\, b \leq x)\\ \exists\, m.\, \forall\, a\, b\, x.\, m\, a\, b \leq a \wedge m\, a\, b \leq b \wedge (x \leq a \wedge x \leq b \longrightarrow x \leq m\, a\, b)\end{array} \tag{67}$$

But these are not difficult to exhibit! Just choose

$$\mathit{join} \equiv \mathit{combine\text{-}matrix}\ \mathit{join},\ \ \mathit{meet} \equiv \mathit{combine\text{-}matrix}\ \mathit{meet}\ . \tag{68}$$

3.6 Positive Part and Negative Part

In lattice-ordered rings (actually in groups, also), both the positive part and the negative part can be defined:

$$\begin{array}{l}\textbf{constdefs}\\ \quad \mathit{pprt} :: \alpha \Rightarrow (\alpha :: \mathit{lordered\text{-}ring})\\ \quad \mathit{pprt}\ x \equiv \mathit{join}\, x\, 0\\ \quad \mathit{nprt} :: \alpha \Rightarrow (\alpha :: \mathit{lordered\text{-}ring})\\ \quad \mathit{nprt}\ x \equiv \mathit{meet}\, x\, 0\end{array} \tag{69}$$

We will write x^+ instead of $pprt\, x$, and x^- instead of $nprt\, x$. We have:

$$0 \le x^+, \quad x^- \le 0, \quad x = x^+ + x^-, \quad x \le y \Longrightarrow x^- \le y^- \wedge x^+ \le y^+ \ . \tag{70}$$

Positive part and negative part come in handy for calculating bounds for a product when bounds for each of the factors of the product are known:

$$\begin{array}{l} [\![a_1 \le a;\, a \le a_2;\, b_1 \le b;\, b \le b_2 \,]\!] \\ \Longrightarrow a * b \le a_2^+ * b_2^+ + a_1^+ * b_2^- + a_2^- * b_1^+ + a_1^- * b_1^- \end{array} \tag{71}$$

In order to prove (71), decompose the factors into their parts and use distributivity. Then take advantage of the monotonicity of positive and negative part:

$$\begin{aligned} a * b &= (a^+ + a^-) * (b^+ + b^-) \\ &= a^+ * b^+ + a^+ * b^- + a^- * b^+ + a^- * b^- \\ &\le a_2^+ * b_2^+ + a_1^+ * b_2^- + a_2^- * b_1^+ + a_1^- * b_1^- \ . \end{aligned}$$

4 The Main Theorem: Proving Bounds by Duality

Now we have everything in place to represent LPs by finite matrices. In sect. 2, we presented the basic idea of how to prove an arbitrarily precise upper bound for the objective function (1) of a given LP. There the LP was represented by matrices whose elements are real numbers:

$$c \in \mathbb{R}^{1\times n}, \quad A \in \mathbb{R}^{m\times n}, \quad b \in \mathbb{R}^{m\times 1} \ , \quad l, u \in \mathbb{R}^{n\times 1}.$$

Dropping the dimensions we arrive at a representation of a real linear program by finite matrices:

$$c, A, b, l, u :: \textit{real matrix} \ .$$

From now on we are always talking in terms of finite matrices.

We need a further modification of our representation of LPs: our method is based on numerical algorithms like the Simplex method, therefore we need to represent the data numerically. We allow for this possibility by looking at *intervals* of linear programs instead of only considering a single LP. Such an interval is given by finite matrices c_1, c_2, A_1, A_2, b, l, u. We can now state the main theorem as it has been proven in Isabelle/HOL:

$$\begin{array}{l} [\![A * x \le b;\, A_1 \le A;\, A \le A_2;\, c_1 \le c;\, c \le c_2;\, l \le x;\, x \le u;\, 0 \le y]\!] \\ \Longrightarrow c * x \le y * b + (\ \text{let } s_1 = c_1 - y * A_2; s_2 = c_2 - y * A_1 \\ \qquad\qquad\qquad\qquad \text{in } s_2^+ * u^+ + s_1^+ * u^- + s_2^- * l^+ + s_1^- * l^-) \ . \end{array} \tag{72}$$

The proof is by standard algebraic manipulations: using $A * x \le b$ and $y \ge 0$,

$$c * x \ \le \ y * b + (c - y * A) * x$$

follows at once. Then one just has to apply (71) to the product $(c - y * A) * x$. Note that this proof not only works for matrices, but for any lattice-ordered ring. Therefore the main theorem is valid also for lattice-ordered rings!

This is how our method works: First, we calculate the approximate optimal solution y of the dual LP. We know our primal LP only approximately, so we can pass only approximate data to the external LP solver. We could pass for example c_1, A_1, b, l, u. The LP solver will return the certificate y, which is only approximately non-negative. Therefore we replace all negative elements of y by 0. We then plug the known numerical data y, c_1, c_2, A_1, A_2, b, l and u into (72) and simplify the resulting theorem. The simplification will rewrite $0 \leq y$ to *True* and the large expression on the right hand side of the inequality to a matrix numeral K with $\mathit{ncols}\, K \leq 1$ and $\mathit{nrows}\, K \leq 1$. The result of our method is therefore the theorem

$$\begin{array}{l}[\![\boldsymbol{A} * \boldsymbol{x} \leq b;\ A_1 \leq \boldsymbol{A};\ \boldsymbol{A} \leq A_2;\ c_1 \leq \boldsymbol{c};\ \boldsymbol{c} \leq c_2;\ l \leq \boldsymbol{x};\ \boldsymbol{x} \leq u]\!] \\ \Longrightarrow \boldsymbol{c} * \boldsymbol{x} \leq K\ .\end{array} \tag{73}$$

In the above theorem, free variables are set in **bold face**. All other identifiers denote matrix numerals.

5 Sparse Matrices and Floats

After reading the previous section, you probably wonder what a matrix numeral might look like. We have chosen to represent matrix numerals in such a way that sparse matrices are encoded efficiently:

types

$$\begin{array}{l}\alpha\ \mathit{spvec} = (\mathit{nat} * \alpha)\ \mathit{list} \\ \alpha\ \mathit{spmat} = (\alpha\ \mathit{spvec})\ \mathit{spvec}\end{array} \tag{74}$$

constdefs

$$\begin{array}{l}\mathit{sparse\text{-}row\text{-}vector} :: \alpha\ \mathit{spvec} \Rightarrow \alpha\ \mathit{matrix} \\ \mathit{sparse\text{-}row\text{-}vector}\ l \equiv \mathit{foldl}\,(\lambda\, m\,(i,e).\ m + (\mathit{singleton\text{-}matrix}\, 0\, i\, e))\, 0\, l \\ \mathit{sparse\text{-}row\text{-}matrix} :: \alpha\ \mathit{spmat} \Rightarrow \alpha\ \mathit{matrix} \\ \mathit{sparse\text{-}row\text{-}matrix}\ L \equiv \\ \quad \mathit{foldl}\,(\lambda\, m\,(j,l).\ m + (\mathit{move\text{-}matrix}\,(\mathit{sparse\text{-}row\text{-}vector}\, l)\, j\, 0))\, 0\, L\end{array} \tag{75}$$

Here $\mathit{singleton\text{-}matrix}\, j\, i\, e$ denotes the matrix whose elements are all zero except the element in row j and column i, which equals e. Furthermore $\mathit{move\text{-}matrix}\, A\, j\, i$ denotes the matrix that one gets if one moves the matrix A by j rows down and i columns right, and fills up the first j rows and i columns with zero elements.

Real numbers are represented as binary, arbitrary precision floating point numbers:

constdef

$$\begin{array}{l}\mathit{float} :: (\mathit{int} * \mathit{int}) \Rightarrow \mathit{real} \\ \mathit{float}\ (m,e) \equiv (\mathit{real}\, m) * 2^e\end{array} \tag{76}$$

Finally, here is an example of a matrix numeral with floats as its elements:

$$[(1, [(1, \mathit{float}\,(7,0)), (3, \mathit{float}\,(-3,2))]), (2, [(0, \mathit{float}\,(1,-3)), (1, \mathit{float}\,(-3,-4))])]$$

$$\xrightarrow{\mathit{sparse\text{-}row\text{-}matrix}} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & -12 \\ \frac{1}{8} & -\frac{3}{16} & 0 & 0 \end{pmatrix}$$

We have formalized addition, subtraction, multiplication, comparison, positive part and negative part directly on sparse vectors and matrices by recursion on lists. The multiplication algorithm for sparse matrices is inspired by the one given in [8].

These operations on sparse vectors/matrices can be proven correct with respect to their finite matrices counterpart via the *sparse-row-matrix* morphism, assuming certain sortedness constraints. This is actually not too hard: all students of an introductory Isabelle/HOL class taught at Technische Universität München have been able to complete these proofs within four weeks as their final assignment with varying help from their tutors. Using these correctness results, one can then easily prove a sparse version of (72).

6 Conclusion

We have presented a novel way to prove arbitrarily precise bounds within higher-order logic for real linear programs that have a priori bounds. Our approach has three main virtues:

1. It is fast. The actual work is done by an external LP solver, the theorem prover has only to check a small certificate. Using a rewriting oracle that is based on the ideas found in [9], this check can be performed so quickly that it is projected that the linear programs arising in the proof of the Kepler conjecture can be bounded in about 10 days on a 3Ghz Pentium 4. The original computer programs from the 1998 proof of the Kepler conjecture that do not generate proofs at all needed back then about 7 days.
2. It decouples reasoning from computing issues. The new notion of finite matrices has been introduced, and it turned out that finite matrices and lattice-ordered rings are a natural choice to describe and reason about our method. At the same time, well-known data structures like sparse matrices can still be used for efficient computing.
3. It is independent from the actual method of solving LPs. This method could be Simplex, but does not have to be.

References

1. Thomas C. Hales. Some algorithms arising in the proof of the Kepler conjecture, sect. 3.1.1., arXiv:math.MG/0205209
2. Thomas C. Hales. A Proof of the Kepler Conjecture, *Annals of Mathematics*, to appear.
3. The Flyspeck Project Fact Sheet.`http://www.math.pitt.edu/~thales/flyspeck/index.html`
4. Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Springer 2002
5. Alexander Schrijver. *Theory of Linear and Integer Programming*, Wiley & Sons 1986

6. Lawrence C. Paulson. Organizing Numerical Theories Using Axiomatic Type Classes. *Journal of Automated Reasoning*, in press.
7. Lawrence C. Paulson. Defining Functions on Equivalence Classes. *ACM Transactions on Computational Logic*, in press.
8. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Transactions on Mathematical Software*, Vol 4. No 3, pp. 250-269, 1978.
9. Bruno Barras. Programming and Computing in HOL. *TPHOLs 2000*, LNCS 1869, pp. 17-37, Springer 2000.
10. Bart Jacobs, Tom Melham. Translating Dependent Type Theory into Higher Order Logic. *Typed Lambda Calculi and Applications: Proceedings of the International Conference*, LNCS 664, pp. 209-229, Springer 1993.
11. Serge Lang. *Algebra*, Addison-Wesley 1974.
12. Garrett Birkhoff. *Lattice Theory*, AMS 1967.
13. László Fuchs. *Partially ordered algebraic systems*, Addison-Wesley 1963.

Essential Incompleteness of Arithmetic Verified by Coq

Russell O'Connor

[1] Institute for Computing and Information Science,
Faculty of Science, Radboud University Nijmegen
[2] The Group in Logic and the Methodology of Science,
University of California, Berkeley
r.oconnor@cs.ru.nl

Abstract. A constructive proof of the Gödel-Rosser incompleteness theorem [9] has been completed using the Coq proof assistant. Some theory of classical first-order logic over an arbitrary language is formalized. A development of primitive recursive functions is given, and all primitive recursive functions are proved to be representable in a weak axiom system. Formulas and proofs are encoded as natural numbers, and functions operating on these codes are proved to be primitive recursive. The weak axiom system is proved to be essentially incomplete. In particular, Peano arithmetic is proved to be consistent in Coq's type theory and therefore is incomplete.

1 Introduction

The Gödel-Rosser incompleteness theorem for arithmetic states that any complete first-order theory of a nice axiom system, using only the symbols $+$, $\times$, $\mathbf{0}$, $\boldsymbol{S}$, and $<$ is inconsistent. A nice axiom system must contain the nine specific axioms of a system called NN. These nine axioms serve to define the previous symbols. A nice axiom system must also be expressible in itself. This last restriction prevents the incompleteness theorem from applying to axioms systems such as the true first order statements about $\mathbb{N}$.

A computer verified proof of Gödel's incompleteness theorem is not new. In 1986 Shankar created a proof of the incompleteness of Z2, hereditarily finite set theory, in the Boyer-Moore theorem prover [11]. My work is the first computer verified proof of the essential incompleteness of arithmetic. Harrison recently completed a proof in HOL Light [6] of the essential incompleteness of Σ_1-complete theories, but has not shown that any particular theory is Σ_1-complete. His work will be included in the next release of HOL Light.

My proof was developed and checked in Coq 7.3.1 using Proof General under XEmacs. It is part of the user contributions to Coq and can be checked in Coq 8.0 [14]. Examples of source code in this document use the new Coq 8.0 notation.

Coq is an implementation of the calculus of (co)inductive constructions. This dependent type theory has intensional equality and is constructive, so my proof is

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 245–260, 2005.

constructive. Actually the proof depends on the Ensembles library which declares an axiom of extensionality for `Ensembles`, but this axiom is never used.

This document points out some of the more interesting problems I encountered when formalizing the incompleteness theorem. My proof mostly follows the presentation of incompleteness given in *An Introduction to Mathematical Logic* [10]. I referred to the supplementary text for the book *Logic for Mathematics and Computer Science* [1] to construct Gödel's β-function. I also use part of Caprotti and Oostdijk's contribution of Pocklington's criterion [2] to prove the Chinese remainder theorem.

This document is organized as follows. First I discuss the difficulties I had when formalizing classical first-order logic over an arbitrary language. This is followed by the definition of a language LNN and an axiom system called NN. Next I give the statement of the essential incompleteness of NN. Then I briefly discuss coding formulas and proofs as natural numbers. Next I discuss primitive recursive functions and the problems I encountered when trying to prove that substitution can be computed by a primitive recursive function. Finally I briefly discuss the fixed-point theorem, Rosser's incompleteness theorem, and the incompleteness of PA. At the end I give some remarks about how to extend my work in order to formalize Gödel's second incompleteness theorem.

1.1 Coq Notation

For those not familiar with Coq syntax, here is a short list of notation

- `->`, `/\`, `\/`, and `~` are the logical connectives $\Rightarrow$, $\wedge$, $\vee$, and $\neg$.
- `A -> B`, `A * B`, and `A + B` form function types, Cartesian product types, and disjoint union types.
- `*`, `+`, and `S` are the arithmetic operations of multiplication, addition, and successor.
- `inl` and `inr` are the left and right injection functions of types `A -> A + B` and `B -> A + B`.
- `::`, and `++` are the list operations cons, and append.
- `_` is an omitted parameter that Coq can infer itself.

For more details see the Coq 8.0 reference manual [14].

2 First-Order Classical Logic

I began by developing the theory of first order classical logic inside Coq. In essence Coq's logic is a formal metalogic to reason about this internal logic.

2.1 Definition of `Language`

I immediately took advantage of Coq's dependent type system by defining `Language` to be a dependent record of types for symbols and an arity function from symbols to $\mathbb{N}$. The Coq code is:

```
Record Language : Type := language
  {Relations : Set;
   Functions : Set;
   arity : Relations + Functions -> nat}.
```

In retrospect it would have been slightly more convenient to use two arity functions instead of using the disjoint union type.

This approach differs from Harrison's definition of first order terms and formulas in HOL Light [5] because HOL Light does not have dependent types. Dependent types allow the type system to enforce that all terms and formulas of a given language are well formed.

2.2 Definition of `Term`

For any given language, a `Term` is either a variable indexed by a natural number or a function symbol plus a list of n terms where n is the arity of the function symbol. My first attempt at writing this in Coq failed.

```
Variable L : Language.
(* Invalid definition *)
Inductive Term0 : Set :=
  | var0 : nat -> Term0
  | apply0 : forall (f : Functions L) (l : List Term0),
       (arity L (inr _ f))=(length l) -> Term0.
```

The type `(arity L (inr _ f))=(length l)` fails to meet Coq's positivity requirement for inductive types. Expanding the definition of `length` reveals a hidden occurrence of `Term0` which is passed as an implicit argument to `length`. It is this occurrence that violates the positivity requirement.

My second attempt met the positivity requirement, but it had other difficulties. A common way to create a polymorphic lists of length `n` is:

```
Inductive Vector (A : Set) : nat -> Set :=
  | Vnil : Vector A 0
  | Vcons : forall (a : A) (n : nat),
       Vector A n -> Vector A (S n).
```

Using this I could have defined `Term` like:

```
Variable L : Language.

Inductive Term1 : Set :=
  | var1 : nat -> Term1
  | apply1 : forall f : Functions L,
       (Vector Term1 (arity L (inr _ f))) -> Term1.
```

My difficulty with this definition was that the induction principle generated by Coq is too weak to work with.

Instead I created two mutually inductive types: `Term` and `Terms`.

```
Variable L : Language.

Inductive Term : Set :=
  | var : nat -> Term
  | apply : forall f : Functions L,
       Terms (arity L (inr _ f)) -> Term
with Terms : nat -> Set :=
  | Tnil : Terms 0
  | Tcons : forall n : nat,
       Term -> Terms n -> Terms (S n).
```

Again the automatically generated induction principle is too weak, so I used the `Scheme` command to generate suitable mutual-inductive principles.

The disadvantage of this approach is that useful lemmas about `Vector`s must be reproved for `Term`s. Some of these lemmas are quite tricky to prove because of the dependent type. For example, proving `forall x : Terms 0, Tnil = x` is not easy.

Recently, Marche has shown me that the `Term1` definition would be adequate. One can explicitly make a sufficient induction principle by using nested `Fixpoint` functions [7].

2.3 Definition of `Formula`

The definition of `Formula` was straightforward.

```
Inductive Formula : Set :=
  | equal : Term -> Term -> Formula
  | atomic : forall r : Relations L, Terms (arity L (inl _ r)) ->
                Formula
  | impH : Formula -> Formula -> Formula
  | notH : Formula -> Formula
  | forallH : nat -> Formula -> Formula.
```

I defined the other logical connectives in terms of `impH`, `notH`, and `forallH`.

The `H` at the end of the logic connectives (such as `impH`) stands for "Hilbert" and is used to distinguish them from Coq's connectives.

For example, the formula $\neg\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.\boldsymbol{x}_0 = \boldsymbol{x}_1$ would be represented by:

```
notH (forallH 0 (forallH 1 (equal (var 0) (var 1))))
```

It would be nice to use higher order abstract syntax to handle bound variables by giving `forallH` the type `(Term -> Formula) -> Formula`. I would represent the above example as:

```
notH (forallH (fun x : Term =>
        (forallH (fun y : Term => (equal x y)))))
```

This technique would require addition work to disallow "exotic terms" that are created by passing a function into `forallH` that does a case analysis on the term

and returning entirely different formulas in different cases. Despeyroux et al. [3] address this problem by creating a complicated predicate that only valid formulas satisfy.

Another choice would have been to use de Bruijn indexes to eliminate named variables. However dealing with free and bound variables with de Bruijn indexes can be difficult.

Using named variables allowed me to closely follow Hodel's work [10]. Also, in order to help persuade people that the statement of the incompleteness theorem is correct, it is helpful to make the underlying definitions as familiar as possible.

Renaming bound variables turned out to be a constant source of work during development because variable names and terms were almost always abstract. In principle the variable names could conflict, so it was constantly necessary to consider this case and deal with it by renaming a bound variable to a fresh one. Perhaps it would have been better to use de Bruijn indexes and a deduction system that only deduced closed formulas.

2.4 Definition of `substituteFormula`

I defined the function `substituteFormula` to substitute a term for all occurrences of a free variable inside a given formula. While the definition of `substituteTerm` is simple structural recursion, substitution for formulas is complicated by quantifiers. Suppose we want to substitute the term s for $\boldsymbol{x}_i$ in the formula $\forall \boldsymbol{x}_j.\varphi$ and $i \neq j$. Suppose $\boldsymbol{x}_j$ is a free variable of s. If we naïvely perform the substitution then the occurrences of $\boldsymbol{x}_j$ in s get captured by the quantifier. One common solution to this problem is to disallow substitution for a term s when s is not *substitutable* for x_i in φ. The solution I take is to rename the bound variable in this case.

$$(\forall \boldsymbol{x}_j.\varphi)[\boldsymbol{x}_i/s] \stackrel{\text{def}}{=} \forall \boldsymbol{x}_k.(\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k])[\boldsymbol{x}_i/s] \text{ where } k \neq i \text{ and } \boldsymbol{x}_k \text{ is not free in } \varphi \text{ or } s$$

Unfortunately this definition is not structurally recursive. The second substitution operates on the result of the first substitution, which is not structurally smaller than the original formula.

Coq will not accept this recursive definition as is; it is necessary to prove the recursion will terminate. I proved that substitution preserves the *depth* of a formula, and that each recursive call operates on a formula of smaller depth.

One of McBride's mantras says, "If my recursion is not structural, I am using the wrong structure" [8, p. 241]. In this case, my recursion is not structural because I am using the wrong recursion. Stoughton shows that it is easier to define substitution that substitutes all variables simultaneously because the recursion is structural [13]. If I had made this definition first, I could have defined substitution of one variable in terms of it and many of my difficulties would have disappeared.

2.5 Definition of `Prf`

I defined the inductive type `(Prf Gamma phi)` to be the type of proofs of `phi`, from the list of assumptions `Gamma`.

```
Inductive Prf : Formulas -> Formula -> Set :=
  | AXM : forall A : Formula, Prf (A :: nil) A
  | MP : forall (Axm1 Axm2 : Formulas) (A B : Formula),
      Prf Axm1 (impH A B) -> Prf Axm2 A ->
         Prf (Axm1 ++ Axm2) B
  | GEN : forall (Axm : Formulas) (A : Formula) (v : nat),
      ~ In v (freeVarListFormula L Axm) -> Prf Axm A ->
         Prf Axm (forallH v A)
  | IMP1 : forall A B : Formula, Prf nil (impH A (impH B A))
  | IMP2 : forall A B C : Formula,
      Prf nil (impH (impH A (impH B C))
                    (impH (impH A B) (impH A C)))
  | CP : forall A B : Formula,
      Prf nil (impH (impH (notH A) (notH B)) (impH B A))
  | FA1 : forall (A : Formula) (v : nat) (t : Term),
      Prf nil (impH (forallH v A) (substituteFormula L A v t))
  | FA2 : forall (A : Formula) (v : nat),
      ~ In v (freeVarFormula L A) ->
         Prf nil (impH A (forallH v A))
  | FA3 : forall (A B : Formula) (v : nat),
      Prf nil
        (impH (forallH v (impH A B))
           (impH (forallH v A) (forallH v B)))
  | EQ1 : Prf nil (equal (var 0) (var 0))
  | EQ2 : Prf nil (impH (equal (var 0) (var 1))
                        (equal (var 1) (var 0)))
  | EQ3 : Prf nil
        (impH (equal (var 0) (var 1))
           (impH (equal (var 1) (var 2)) (equal (var 0) (var 2))))
  | EQ4 : forall R : Relations L, Prf nil (AxmEq4 R)
  | EQ5 : forall f : Functions L, Prf nil (AxmEq5 f).
```

`AxmEq4` and `AxmEq5` are recursive functions that generate the equality axioms for relations and functions. `AxmEq4 R` generates

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 \Rightarrow \ldots \Rightarrow \boldsymbol{x}_{2n-2} = \boldsymbol{x}_{2n-1} \Rightarrow (R(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{2n-2}) \Leftrightarrow R(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{2n-1}))$$

and `AxmEq5 f` generates

$$\boldsymbol{x}_0 = \boldsymbol{x}_1 \Rightarrow \ldots \Rightarrow \boldsymbol{x}_{2n-2} = \boldsymbol{x}_{2n-1} \Rightarrow f(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{2n-2}) = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{2n-1})$$

I found that replacing ellipses from informal proofs with recursive functions was one of the most difficult tasks. The informal proof does not contain information on what inductive hypothesis should be used when reasoning about these recursive definitions. Figuring out the correct inductive hypotheses was not easy.

2.6 Definition of `SysPrf`

There are some problems with the definition of `Prf` given. It requires the list of axioms to be in the correct order for the proof. For example, if we have `Prf Gamma1 (impH phi psi)` and `Prf Gamma2 phi` then we can conclude only `Prf Gamma1++Gamma2 psi`. We cannot conclude `Prf Gamma2++Gamma1 psi` or any other permutation of `psi`. If an axiom is used more than once, it must appear in the list more than once. If an axiom is never used, it must not appear. Also, the number of axioms must be finite because they form a list.

To solve this problem, I defined a `System` to be `Ensemble Formula`, and `(SysPrf T phi)` to be the proposition that the system `T` proves `phi`.

```
Definition System := Ensemble Formula.
Definition mem := Ensembles.In.

Definition SysPrf (T : System) (f : Formula) :=
  exists Axm : Formulas,
    (exists prf : Prf Axm f,
       (forall g : Formula, In g Axm -> mem _ T g)).
```

`Ensemble A` represents subsets of `A` by the functions `A -> Prop`. `a : A` is considered to be a member of `T : Ensemble A` if and only if the type `T a` is inhabited. I also defined `mem` to be `Ensembles.In` so that it does not conflict with `List.In`.

2.7 The Deduction Theorem

The deduction theorem states that if $\Gamma \cup \{\varphi\} \vdash \psi$ then $\Gamma \vdash \varphi \Rightarrow \psi$.

There is a choice of whether the side condition for the $\forall$-generalization rule, `~ In v (freeVarListFormula L Axm)`, should be required or not. If this side condition is removed then the deduction theorem requires a side condition on it. Usually all the formulas in an axiom system are closed, so the side condition on the $\forall$-generalization is easy to show. So I decided to keep the side condition on the $\forall$-generalization rule.

At one point the proof of the deduction theorem requires proving that if $\Gamma \cup \{\varphi\} \vdash \psi$ because $\psi \in \Gamma \cup \{\varphi\}$, then $\Gamma \vdash \varphi \Rightarrow \psi$. There are two cases to consider. If $\psi = \varphi$ then the result easily follows from the reflexivity of $\Rightarrow$. Otherwise $\psi \in \Gamma$, and therefore $\Gamma \vdash \psi$. The result then follows. In order to constructively make this choice it is necessary to decide whether $\psi = \varphi$ or not. This requires `Formula` to be a decidable type, and that requires the language `L` to be decidable. Since `L` could be anything, I needed to add hypotheses that the function and relation symbols are decidable types.

```
- forall x y : Functions L, { x=y } + { x<>y }
- forall x y : Relations L, { x=y } + { x<>y }.
```

I used the deduction theorem without restriction and ended up using the hypotheses in many lemmas. I expect that many of these lemmas could be proved without assuming the decidability of the language. It is hard to imagine a useful language that is not decidable, so I do not feel too bad about using these hypotheses in unnecessary places.

2.8 Languages and Theories of Number Theory

I created two languages. The first language, `LNT`, is the language of number theory and just has the function symbols `Plus`, `Times`, `Succ`, and `Zero` with appropriate arities. The second language, `LNN`, is the language of NN and has the same function symbols as `LNT` plus one relation symbol for less than, `LT`.

I define two axiom systems: `NN` and `PA`. `NN` and `PA` share six axioms.

1. $\forall \boldsymbol{x}_0.\neg \boldsymbol{S}\boldsymbol{x}_0 = \boldsymbol{0}$
2. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.(\boldsymbol{S}\boldsymbol{x}_0 = \boldsymbol{S}\boldsymbol{x}_1 \Rightarrow \boldsymbol{x}_0 = \boldsymbol{x}_1)$
3. $\forall \boldsymbol{x}_0.\boldsymbol{x}_0 + \boldsymbol{0} = \boldsymbol{x}_0$
4. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.\boldsymbol{x}_0 + \boldsymbol{S}\boldsymbol{x}_1 = \boldsymbol{S}(\boldsymbol{x}_0 + \boldsymbol{x}_1)$
5. $\forall \boldsymbol{x}_0.\boldsymbol{x}_0 \times \boldsymbol{0} = \boldsymbol{0}$
6. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.\boldsymbol{x}_0 \times \boldsymbol{S}\boldsymbol{x}_1 = (\boldsymbol{x}_0 \times \boldsymbol{x}_1) + \boldsymbol{x}_0$

`NN` has three additional axioms about less than.

1. $\forall \boldsymbol{x}_0.\neg \boldsymbol{x}_0 < \boldsymbol{0}$
2. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.(\boldsymbol{x}_0 < \boldsymbol{S}\boldsymbol{x}_1 \Rightarrow (\boldsymbol{x}_0 = \boldsymbol{x}_1 \vee \boldsymbol{x}_0 < \boldsymbol{x}_1))$
3. $\forall \boldsymbol{x}_0.\forall \boldsymbol{x}_1.(\boldsymbol{x}_0 < \boldsymbol{x}_1 \vee \boldsymbol{x}_0 = \boldsymbol{x}_1 \vee \boldsymbol{x}_1 < \boldsymbol{x}_0)$

`PA` has an infinite number of induction axioms that follow one schema.

1. (schema) $\forall \boldsymbol{x}_{i_1}. \ldots \forall \boldsymbol{x}_{i_n}.\varphi[\boldsymbol{x}_j/\boldsymbol{0}] \Rightarrow \forall \boldsymbol{x}_j.(\varphi \Rightarrow \varphi[\boldsymbol{x}_j/\boldsymbol{S}\boldsymbol{x}_j]) \Rightarrow \forall \boldsymbol{x}_j.\varphi$

The $\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{i_n}$ are the free variables of $\forall \boldsymbol{x}_j.\varphi$. The quantifiers ensure that all the axioms of PA are closed.

Because NN is in a different language than PA, a proof in NN is not a proof in PA. In order to reuse the work done in NN, I created a function called `LNN2LNT_formula` to convert formulas in `LNN` into formulas in `LNT` by replacing occurrences of $t_0 < t_1$ with $(\exists \boldsymbol{x}_2.\boldsymbol{x}_0 + (\boldsymbol{S}\boldsymbol{x}_2) = \boldsymbol{x}_1)[\boldsymbol{x}_0/t_0, \boldsymbol{x}_1/t_1]$—$\varphi[\boldsymbol{x}_0/t_0, \boldsymbol{x}_1/t_1]$ is the simultaneous substitution of t_0 for $\boldsymbol{x}_0$ and t_1 for $\boldsymbol{x}_1$. Then I proved that if $\mathrm{NN} \vdash \varphi$ then $\mathrm{PA} \vdash \texttt{LNN2LNT\_formula}(\varphi)$.

I also created the function `natToTerm : nat -> Term` to return the closed term representing a given natural number. In this document I will refer to this function as $\ulcorner \boldsymbol{.} \urcorner$, so $\ulcorner 0 \urcorner = \boldsymbol{0}$, $\ulcorner 1 \urcorner = \boldsymbol{S0}$, etc.

3 Coding

To prove the incompleteness theorem, it is necessary for the inner logic to reason about proofs and formulas, but the inner logic can only reason about natural numbers. It is therefore necessary to code proofs and formulas as natural numbers.

Gödel's original approach was to code a formula as a list of numbers and then code that list using properties from the prime decomposition theorem[4]. I avoided needing theorems about prime decomposition by using the Cantor pairing function instead. The Cantor pairing function, `cPair`, is a commonly used bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

$$\texttt{cPair}(a, b) \stackrel{\text{def}}{=} a + \sum_{i=1}^{a+b} i$$

All my inductive structures were easy to recursively encode. I gave each constructor a unique number and paired that number with the encoding of all its parameters. For example, I defined `codeFormula` as:

```
Fixpoint codeFormula (f : Formula) : nat :=
  match f with
  | fol.equal t1 t2 => cPair 0 (cPair (codeTerm t1) (codeTerm t2))
  | fol.impH f1 f2 =>
       cPair 1 (cPair (codeFormula f1) (codeFormula f2))
  | fol.notH f1 => cPair 2 (codeFormula f1)
  | fol.forallH n f1 => cPair 3 (cPair n (codeFormula f1))
  | fol.atomic R ts => cPair (4+(codeR R)) (codeTerms _ ts)
  end.
```

where `codeR` is a coding of the relation symbols for the language.

I will use $\ulcorner\varphi\urcorner$ for $\ulcorner$`codeFormula` $\varphi\urcorner$ and $\ulcorner t\urcorner$ for $\ulcorner$`codeTerm` $t\urcorner$.

4 The Statement of Incompleteness

The incompleteness theorem states the essential incompleteness of NN, meaning that for every axiom system T such that

- NN $\subseteq T$
- T can represent its own axioms
- T is a decidable set

then there exists a sentence φ such that if $T \vdash \varphi$ or $T \vdash \neg\varphi$ then T is inconsistent.

The theorem is only about proofs in `LNN`, the language of NN. This statement does not show the incompleteness of theories that extend the language.

In Coq the theorem is stated as as:

```
Theorem Incompleteness
     : forall T : System,
       Included Formula NN T ->
       RepresentsInSelf T ->
       DecidableSet Formula T ->
       exists f : Formula,
         Sentence f /\
         (SysPrf T f \/ SysPrf T (notH f) -> Inconsistent LNN T).
```

A `System` is `Inconsistent` if it proves all formulas.

```
Definition Inconsistent (T : System) :=
   forall f : Formula, SysPrf T f.
```

A `Sentence` is a `Formula` without any free variables.

```
Definition Sentence (f : Formula) :=
   forall v : nat, ~ In v (freeVarFormula LNN f).
```

A `DecidableSet` is an `Ensemble` such that every item either belongs to the `Ensemble` or does not belong to the `Ensemble`. This hypothesis is trivially true in classical logic, but in constructive logic I needed it to prove the strong constructive existential quantifier in the statement of incompleteness.

```
Definition DecidableSet (A : Type)(s : Ensemble A) :=
   forall x : A, mem A s x \/ ~ mem A s x.
```

The `RepresentsInSelf` hypothesis restricts what the `System` T can be. The statement of essential incompleteness normally requires T be a recursive set. Instead I use the weaker hypothesis that the set T is expressible in the system T.

Given a system T extending NN and another system U along with a formula φ_U with at most one free variable $\boldsymbol{x}_i$, we say φ_U expresses the axiom system U in T if the following hold for all formulas ψ.

1. if $\psi \in U$ then $T \vdash \varphi_U[\boldsymbol{x}_i/\ulcorner\psi\urcorner]$
2. if $\psi \notin U$ then $T \vdash \neg\varphi_U[\boldsymbol{x}_i/\ulcorner\psi\urcorner]$

U is expressible in T if there exists a formula φ_U such that φ_U expresses the axiom system U in T.

In Coq I write the statement T is expressible in T as

```
Definition RepresentsInSelf (T : System) :=
exists rep : Formula, exists v : nat,
(forall x : nat, In x (freeVarFormula LNN rep) -> x = v)  /\
(forall f : Formula,
        mem Formula T f ->
        SysPrf T (substituteFormula LNN rep v
                    (natToTerm (codeFormula f)))) /\
(forall f : Formula,
        ~ mem Formula T f ->
        SysPrf T (notH (substituteFormula LNN rep v
                          (natToTerm (codeFormula f))))).
```

This is weaker than requiring that T be a recursive set because any recursive set of axioms T is expressible in NN. Since T is an extension of NN, any recursive set of axioms T is expressible in T.

By using this weaker hypothesis I avoid defining what a recursive set is. Also, in this form the theorem could be used to prove that any complete and consistent theory of arithmetic cannot define its own axioms. In particular, this could be used to prove Tarski's theorem that the truth predicate is not definable.

5 Primitive Recursive Functions

A common approach to proving the incompleteness theorem is to prove that every primitive recursive function is representable. Informally an n-ary function f is representable in NN if there exists a formula φ such that

1. the free variables of φ are among $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_n$.
2. for all $a_1, \ldots, a_n : \mathbb{N}$,
   $\mathrm{NN} \vdash (\varphi \Rightarrow \boldsymbol{x}_0 = \ulcorner f(a_1, \ldots, a_n) \urcorner)[\boldsymbol{x}_1/\ulcorner a_1 \urcorner, \ldots, \boldsymbol{x}_n/\ulcorner a_n \urcorner]$

I defined the type `PrimRec n` as:

```
Inductive PrimRec : nat -> Set :=
  | succFunc : PrimRec 1
  | zeroFunc : PrimRec 0
  | projFunc : forall n m : nat, m < n -> PrimRec n
  | composeFunc :
      forall (n m : nat) (g : PrimRecs n m) (h : PrimRec m),
        PrimRec n
  | primRecFunc :
      forall (n : nat) (g : PrimRec n) (h : PrimRec (S (S n))),
        PrimRec (S n)
with PrimRecs : nat -> nat -> Set :=
  | PRnil : forall n : nat, PrimRecs n 0
  | PRcons : forall n m : nat,
      PrimRec n -> PrimRecs n m -> PrimRecs n (S m).
```

`PrimRec n` is the expression of an n-ary primitive recursive function, but it is not itself a function. I defined `evalPrimRec : forall n : nat, PrimRec n -> naryFunc n` to convert the expression into a function. Rather than working directly with primitive recursive expressions, I worked with particular Coq functions and proved they were extensionally equivalent to the evaluation of primitive recursive expressions.

I proved that every primitive recursive function is representable in NN. This required using Gödel's β-function along with the Chinese remainder theorem. The β-function is a function that codes array indexing. A finite list of numbers $a_0, \ldots, a_n$ is coded as a pair of numbers (x, y) and $\beta(x, y, i) = a_i$. The β-function is special because it is defined in terms of plus and times and is non-recursive. The Chinese remainder theorem is used to prove that the β-function works.

I took care to make the formulas representing the primitive recursive functions clearly Σ_1 by ensuring that only the unbounded quantifiers are existential; however, I did not prove that the formulas are Σ_1 because it is not needed for the first incompleteness theorem. Such a proof could be used for the second incompleteness theorem [12].

5.1 `codeSubFormula` Is Primitive Recursive

I proved that substitution is primitive recursive. Since substitution is defined in terms of `Formula` and `Term`, it itself cannot be primitive recursive. Instead I proved that the corresponding function operating on codes is primitive recursive. This function is called `codeSubFormula` and I proved it is correct in the following sense.

$$\texttt{codeSubFormula}(\ulcorner \varphi \urcorner, i, \ulcorner s \urcorner) = \ulcorner \varphi[\boldsymbol{x}_i/s] \urcorner$$

Next I proved that it is primitive recursive. This proof is very difficult. The problem is again with the need to rebind bound variables. Normally one would attempt to create this primitive recursive function by using course-of-values recursion. Course-of-values recursion requires all recursive calls have a smaller code than the original call. Renaming a bound variable requires two recursive calls. Recall the definition of substitution in this case:

$$(\forall \boldsymbol{x}_j.\varphi)[\boldsymbol{x}_i/s] \stackrel{\text{def}}{=} \forall \boldsymbol{x}_k.(\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k])[\boldsymbol{x}_i/s] \text{ where } k \neq i \text{ and } \boldsymbol{x}_k \text{ is not free in } \varphi \text{ or } s$$

If one is lucky one might be able to make the inner recursive call. But there is no reason to suspect the input to the second recursive call, $\varphi[\boldsymbol{x}_j/\boldsymbol{x}_k]$, is going to have a smaller code than the original input, $\forall \boldsymbol{x}_j.\varphi$.

If I had used the alternative definition of substitution, where all variables are substituted simultaneously, there would still be problems. The input would include a list of variable and term pairs. In this case a new pair would be added to the list when making the recursive call, so the input to the recursive call could still have a larger code than the input to the original call.

It seems that using course-of-values recursion is difficult or impossible. Instead I introduce the notion of the trace of the computation of substitution. Think of the trace of computation as a finite tree where the nodes contain the input and output of each recursive call. The subtrees of a node are the traces of the computation of the recursive calls. This tree can be coded as a number. I proved that there is a primitive recursive function that can check to see if a number represents a trace of the computation of substitution.

The key to solving this problem is to create a primitive recursive function that computes a bound on how large the code of the trace of computation can be for a given input. With this I created another primitive recursive function that searches for the trace of computation up to this bound. Once the trace is found—I proved that it must be found—the function extracts the result from the trace and returns it.

5.2 `checkPrf` Is Primitive Recursive

Given a code for a formula and a code for a proof, the function `checkPrf` returns 0 if the proof does not prove the formula, otherwise it returns one plus the code of the list of axioms used in the proof. I proved this function is primitive recursive, as well as proving that it is correct in the sense that for every proof p of φ from a list of axioms Γ, $\texttt{checkPrf}(\ulcorner\varphi\urcorner, \ulcorner p\urcorner) = 1 + \ulcorner\Gamma\urcorner$; and for all $n, m : \mathbb{N}$ if $\texttt{checkPrf}(n, m) \neq 0$ then there exists φ, Γ, and some proof p of φ from Γ such that $\ulcorner\varphi\urcorner = n$ and $\ulcorner p\urcorner = m$.

For any axiom system U expressible in T, I created the formulas `codeSysPrf` and `codeSysPf`. $\texttt{codeSysPrf}[\boldsymbol{x}_0/\ulcorner n\urcorner, \boldsymbol{x}_1/\ulcorner m\urcorner]$ is provable in T if m is the code of a proof in U of a formula coded by n. $\texttt{codeSysPf}[\boldsymbol{x}_0/\ulcorner n\urcorner]$ is provable in T if there exists a proof in U of a formula coded by n.

`codeSysPrf` and `codeSysPf` are not derived from a primitive recursive functions because I wanted to prove the incompleteness of axiom systems that may not have a primitive recursive characteristic function.

6 Fixed Point Theorem and Rosser's Incompleteness Theorem

The fixed point theorem states that for every formula φ there is some formula ψ such that

$$\mathrm{NN} \vdash \psi \Leftrightarrow \varphi[\boldsymbol{x}_i/\ulcorner\psi\urcorner]$$

and that the free variables of ψ are that of φ less $\boldsymbol{x}_i$.

The fixed point theorem allows one to create "self-referential sentences". I used this to create Rosser's sentence which states that for every code of a proof of itself, there is a smaller code of a proof of its negation. The proof of Rosser's incompleteness theorem requires doing a bounded search for a proof, and this requires knowing what is and what is not a proof in the system. For this reason, I require the decidability of the axiom system. Without a decision procedure for the axiom system, I cannot constructively do the search.

6.1 Incompleteness of PA

To demonstrate the incompleteness theorem I used it to prove the incompleteness of PA. I created a primitive recursive predicate for the codes of the axioms of PA. Coq is sufficiently powerful to prove the consistency of PA by proving that the natural numbers model PA.

One subtle point is that Coq's logic is constructive while the internal logic is classical. One cannot interpret a formula of the internal logic directly in Coq and expect it to be provable if it is provable in the internal logic. Instead I use a double negation translation of the formulas. The translated formula will always hold if it holds in the internal logic.

The consistency of PA along with the expressibility of its axioms and the translations of proofs from NN to PA allowed me to apply Rosser's incompleteness theorem and prove the incompleteness of PA—there exists a sentence φ such that neither $\mathrm{PA} \vdash \varphi$ nor $\mathrm{PA} \vdash \neg\varphi$.

```
Theorem PAIncomplete :
 exists f : Formula,
   (forall v : nat, ~ In v (freeVarFormula LNT f)) /\
   ~ (SysPrf PA f \/ SysPrf PA (notH f)).
```

7 Remarks

7.1 Extracting the Sentence

Because my proof is constructive, it is possible, in principle, to compute this sentence that makes PA incomplete. This was not done for two reasons. The first reason is that the existential statement lives in Coq's `Prop` universe, and Coq's only extracts from its `Set` universe. This was an error on my part. I should have used Coq's `Set` existential quantifier; this problem would be fairly easy to fix. The second reason is that the sentence contains a closed term of the code

of most of itself. I believe this code is a very large number and it is written in unary notation. This would likely make the sentence far to large to be actually printed.

7.2 Robinson's System Q

The proof of essential incompleteness is usually carried out for Robinson's system Q. Instead I followed Hodel's development [10] and used NN. Q is PA with the induction schema replaced with $\forall \boldsymbol{x}_0.\exists \boldsymbol{x}_1.(\boldsymbol{x}_0 = \mathbf{0} \vee \boldsymbol{x}_0 = \boldsymbol{S}\boldsymbol{x}_1)$. All of NN axioms are Π_1 whereas Q has the above Π_2 axiom. Both axiom systems are finite.

Neither system is strictly weaker than the other, so it would not be possible to use the essential incompleteness of one to get the essential incompleteness of the other; however both NN and Q are sufficiently powerful to prove a small number of needed lemmas, and afterward only these lemmas are used. If one abstracts my proof at these lemmas, it would then be easy to prove the essential incompleteness of both Q and NN.

7.3 Comparisons with Shankar's 1986 Proof

It is worth noting the differences between this formalization of the incompleteness theorem and Shankar's 1986 proof in the Boyer-Moore theorem prover. The most notable difference is the proof systems. In Coq the user is expected to input the proof, in the form of a proof script, and Coq will check the correctness of the proof. In the Boyer-Moore theorem prover the user states a series of lemmas and the system generates the proofs. However, using the Boyer-Moore proof system requires feeding it a "well-chosen sequence of lemmas" [11, p. xii], so it would seem the information being fed into the two systems is similar.

There are some notable semantic differences between Shankar's statement of incompleteness and mine. His theorem only states that finite extensions of Z2, hereditarily finite set theory, are incomplete, whereas my theorem states that even infinite extensions of NN are incomplete as long as they are self-representable. Also Shankar's internal logic allows axioms to define new relation or function symbols as long as they come with the required proofs of admissibility. Such extensions are conservative over Z2, but no computer verified proof of this fact is given. My internal logic does not allow new symbols. Finally, I prove the essential incompleteness of NN, which is in the language of arithmetic. Without any set structures the proof is somewhat more difficult because it requires using Gödel's β-function.

One of Shankar's goals when creating his proof was to use a proof system without modifications. Unfortunately he was not able to meet that goal; he ended up making some improvements to the Boyer-Moore theorem prover. My proof was developed in Coq without any modifications.

7.4 Gödel's Second Incompleteness Theorem

The second incompleteness theorem states that if T is a recursive system extending PA—actually a weaker system could be used here—and $T \vdash \mathrm{Con}_T$ then T is

inconsistent. Con_T is some reasonable formula stating the consistency of T, such as $\neg \mathrm{Pr}_T(\ulcorner \mathbf{0} = \boldsymbol{S}\mathbf{0} \urcorner)$, where Pr_T is the provability predicate `codeSysPf` for T.

If I had created a formal proof in PA, I would have $\vdash_{\mathrm{PA}}$ "Gödel's first incompleteness theorem". This could then be mechanically transformed to create another formal proof in PA that $\vdash_{\mathrm{PA}}$ (PA $\vdash$ "Gödel's first incompleteness theorem"). The reader can verify that the second incompleteness theorem follows from this. Unfortunately I have only shown that $\vdash_{Coq}$ "Gödel's first incompleteness theorem", so the above argument cannot be used to create a proof of the second incompleteness theorem.

Still, this work can be used as a basis for formalizing the second incompleteness theorem. The approach would be to formalize the Hilbert-Bernays-Löb derivability conditions:

1. if PA $\vdash \varphi$ then PA $\vdash \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner)$
2. PA $\vdash \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner) \Rightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner) \urcorner)$
3. PA $\vdash \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \Rightarrow \psi \urcorner) \Rightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner) \Rightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \psi \urcorner)$

The second condition is the most difficult to prove. It is usually proved by first proving that for every Σ_1 sentence φ, PA $\vdash \varphi \Rightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \varphi \urcorner)$. Because I made sure that all primitive recursive functions are representable by a Σ_1 formula, it would be easy to go from this theorem to the second Hilbert-Bernays-Löb condition.

8 Statistics

My proof, excluding standard libraries and the library for Pocklington's criterion [2], consists of 46 source files, 7 036 lines of specifications, 37 906 lines of proof, and 1 267 747 total characters. The size of the gzipped tarball (`gzip -9`) of all the source files is 146 008 bytes, which is an estimate of the information content of my proof.

Acknowledgements

I would like to thank NSERC for providing funding for this research. I thank Robert Schneck for introducing me to Coq, and helping me out at the beginning. I would like to thank Nikita Borisov for letting me use his computer when the proof became to large for my poor laptop. I would also like to thank my Berkeley advisor, Leo Harrington, for his advice on improving upon Hodel's proof. And last, but not least, thanks to the Coq team, because without Coq there would be no proof.

References

1. Stanley N. Burris. Logic for mathematics and computer science: Supplementary text. `http://www.math.uwaterloo.ca/~snburris/htdocs/LOGIC/stext.html`, 1997.

2. Olga Caprotti and Martijn Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *J. Symb. Comput.*, 32(1/2):55–70, 2001.
3. Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in coq. In *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, London, UK, 1994. Springer-Verlag.
4. K. Gödel. Ueber Formal Unentscheidbare sätze der Principia Mathematica und Verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. english translation: On Formally Undecidable Propositions of Principia Mathematica and Related Systems I, Oliver & Boyd, London, 1962.
5. John Harrison. Formalizing basic first order model theory. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lecture Notes in Computer Science*, pages 153–170, Canberra, Australia, 1998. Springer-Verlag.
6. John Harrison. The HOL-Light manual, 2000.
7. Claude Marche. Fwd: Question about fixpoint. Coq club mailing list correspondence, `http://pauillac.inria.fr/pipermail/coq-club/2005/001641.html`, [cited 2005-02-07], February 2005.
8. Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.
9. Russell O'Connor. The Gödel-Rosser 1st incompleteness theorem. `http://r6.ca/Goedel20050512.tar.gz`, March 2005.
10. Richard E. Hodel. *An Introduction to Mathematical Logic.* PWS Pub. Co., 1995.
11. N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1994.
12. J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.
13. Allen Stoughton. Substitution revisited. 59(3):317–325, August 1988.
14. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

Verification of BDD Normalization

Veronika Ortner and Norbert Schirmer

Technische Universität München, Institut für Informatik
{ortner, schirmer}@in.tum.de

Abstract. We present the verification of the normalization of a binary decision diagram (BDD). The normalization follows the original algorithm presented by Bryant in 1986 and transforms an ordered BDD in a reduced, ordered and shared BDD. The verification is based on Hoare logics and is carried out in the theorem prover Isabelle/HOL. The work is both a case study for verification of procedures on a complex pointer structure, as well as interesting on its own, since it is the first proof of functional correctness of the pointer based normalization process we are aware of.

1 Introduction

Binary Decision Diagrams (BDDs) are a canonical, memory efficient pointer structure to represent boolean functions, with a wide spread application in computer science. They had a decisive impact on scaling up the technology of model checking to large state-spaces to handle practical applications [5]. BDDs were introduced by Bryant [3], and later on their implementation was refined [2]. The efficiency of the BDD algorithms stems from the sharing of subgraphs in the BDD. Some properties of the BDD, like the uniqueness of the representation of a function, rely on the fact that the graph is maximally shared. So the algorithms manipulating BDDs have to ensure this invariant. However the formal verification of the algorithms has never received great attention. We are only aware of the work of Verma et al. [14] and of Kristic and Matthews [7]. Maybe one reason for the lack of formal verification is the problem of reasoning about imperative pointer programs in general, which is still an area of active research, recently in two main directions: The integration and mechanization of pointer verification in theorem provers [1,8,6], and the development of a new logic, namely separation logic [12]. In this context our work contributes to two aspects. On the one hand it presents the first formal verification of the pointer based normalization algorithm for BDDs as presented by Bryant. Verma et al. [14] and also Kristic and Matthews [7] use a different (more abstract) model of BDDs, where the normalization algorithm is no issue, since they ensure that only normalized BDDs will be constructed at all. Although modern BDD packages also follow this approach, and avoid the costly normalization process, the concepts we introduce to formally describe the invariants on the BDD pointer structure can also serve as basis in a more involved setting like [2]. On the other hand this work is a case study on the feasibility of pointer verification based on Hoare logics in a theorem

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 261–277, 2005.

prover. It carries on the approach of [8] to (recursive) procedures. In contrast to separation logic, which is difficult to combine with existing theorem proving infrastructure, our embedding of Hoare logics fits seamlessly into Isabelle/HOL.

The rest of the paper is structured as follows. In Sect. 2 we give a short introduction to Isabelle/HOL in general and the Hoare logic module, that we will use for our verification work. Sect. 3 gives an informal overview of BDDs and Sect. 4 introduces our formalization of them. Sect. 5 is devoted to the normalization of BDDs, where we explain the algorithm and describe the assertions and invariants that we have used for the correctness proof. Finally Sect. 6 concludes.

2 Preliminaries

2.1 Isabelle

Isabelle/HOL [10] is an interactive theorem prover for HOL, higher order logic, with facilities for defining data types, records, inductive sets as well as primitive and total general recursive functions. Most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight some of Isabelle's nonstandard notation.

There are the usual type constructors $T_1 \times T_2$ for product and $T_1 \Rightarrow T_2$ for function space. To emulate partial functions the polymorphic option type is frequently used: **datatype** *'a option* = *None* | *Some 'a*. Here *'a* is a type variable, *None* stands for the undefined value and *Some x* for a defined value *x*. A partial function from type T_1 to type T_2 can be modelled as $T_1 \Rightarrow (T_2$ *option*). Lists (type *'a list*) come with the empty list [], the infix constructor # and the infix @ that appends two lists, and the conversion function *set* from lists to sets. The nth element of a list *xs* is obtained by *xs* ! *n*. The standard function *map* is also available.

2.2 The Hoare Logic Module

Before considering the algorithm of BDD normalization in detail, we first take a brief look at the verification environment for imperative programs [13] built on top of Isabelle/HOL. It embeds a sequential imperative programming language with (mutually) recursive procedures, local and global variables and a heap for pointer structures into Isabelle/HOL. The expressions used in the language are modelled as ordinary HOL expressions. Therefore both "pseudo code" and C programs can be expressed in a uniform framework, giving the user the freedom to choose the proper abstraction level.

To define the state-space, we use **record**s in Isabelle/HOL [10], which contain every program variable used in the implementation. We can refer to these state-space components by specifying $\acute{v}$ for the current state of the component *v* and ${}^{\sigma}v$ for the same component at a fixed state σ. Assertions are sets of states and we provide special brackets $\{\!|\ |\!\}$ for them, e.g. $\{\!|M = 2|\!\}$ is a shorthand for the ordinary set comprehension $\{s \mid M\, s = 2\}$, which is the set of states where variable *M* is equal to *2*. *M* is a record selector of the state-space.

A judgement in our Hoare logic is of the general form $\Gamma,\Theta\vdash P\ c\ Q$ for partial correctness and $\Gamma,\Theta\vdash_t P\ c\ Q$ for total correctness, where P and Q figure as pre- and postcondition. The two remaining variables are premises of the Hoare triple, Γ being the procedure environment and context Θ representing a set of Hoare triples that we may assume. The procedure environment maps procedure names to their bodies. The Hoare triples in Θ are important for proving recursive procedures. An empty set of assumptions can be omitted.

Moreover the module supplies a verification condition generator built on top of the Hoare logic for the programming language.

3 Binary Decision Diagrams

"Many problems in digital logic design and testing, artificial intelligence, and combinatorics can be expressed as a sequence of operations on Boolean functions." ([3], p. 1) Thus the representation of Boolean functions by an efficient data-structure is of high interest for computer science. *Binary Decision Diagrams*, which represent the underlying binary decision tree of a Boolean function as a directed acyclic graph (DAG), save storage space and computation time by eliminating redundancy from the canonical representations. Besides, using *reduced*, *ordered* and *shared* BDDs allows us to provide a unique representation for each function. An inner node of the BDD contains a variable over which the Boolean function is defined, together with a pointer to the left and right sub-BDD. Given a valuation of the variables the value of the Boolean function encoded in the BDD is obtained by traversing the BDD according to the valuation of the variables. The leaf that we reach holds the value of the function under the given valuation.

As BDDs are an efficient representation for Boolean functions, which are used in a lot of domains of computer science, there is a wide variety of imaginable operations on them. We will only treat the normalization here, which has an ordered BDD as its argument and removes all redundancies contained in it. The result is an ordered, reduced and shared BDD implementing the same Boolean function. The normalization follows the algorithm presented in [3], where it is used as a central building block for further BDD-algorithms.

The basic transformations on the BDD that occur during normalization are *reducing* and *sharing* (Fig. 1). A node is *reduced* from the BDD if it is irrelevant for the encoded Boolean function: if the left and the right child both point to the same node, it is irrelevant if we choose to go left or right during evaluation. Two sub-BDDs are *shared* if they contain the same decision tree. Note that sharing does not change the structure of the underlying decision tree, but only the structure of the DAG, whereas reducing also changes the decision tree. None of the transformations change the encoded Boolean function.

BDDs are an extreme area of application for pointer programs with operations involving side effects due to the high degree of sharing in the data-structure. Because of their efficiency in computation time and storage, they are highly popular for the processing of Boolean functions. Altogether BDDs constitute a

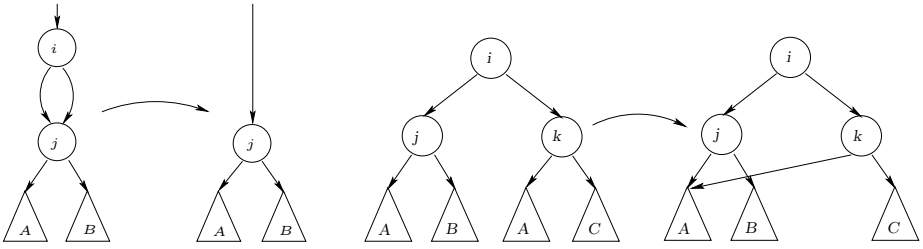


Fig. 1. Illustration of normalization operations

perfect domain for a case study for our Hoare logic. They represent a practically relevant subject and include the important pointer program features, which pose problems for verification.

4 Formalization of BDDs

4.1 State-Space

In order to represent all variables and the heap used in the program we use Isabelle records. Our model of the heap follows Burstall's [4] idea, recently emphasized by Bornat [1] and Mehta and Nipkow [8]: we have one heap f of type $ref \Rightarrow value$ for each component f of type *value* of a BDD-node. Type *ref* is our abstract view on addresses. It is isomorphic to the natural numbers and contains *Null* as an element. Figure 2 shows the C-style structure for a DAG node and the corresponding *Isabelle* records we use to represent the state-space. A C-like selection `p->var` becomes function application *var p* in our model. The global components (in our case the split heap) are grouped together in one field *globals* of the state-space-record. The remaining fields are used for local variables and procedure parameters. The semantics of our programming language model is defined via updates on this state-space [13]. The separation of global and local components is used to handle procedure calls.

Every BDD-node contains a variable *var* which is encoded as a natural number. We reserve the variables 0 and 1 for the terminal nodes. Besides every node

```
struct node {
 nat var;
 struct node* left;
 struct node* right;
 struct node* rep;
 struct node* next;
 bool mark;
};
```

record *heap* =
  var :: *ref* ⇒ *nat*
  left :: *ref* ⇒ *ref*
  right :: *ref* ⇒ *ref*
  rep :: *ref* ⇒ *ref*
  next :: *ref* ⇒ *ref*
  mark :: *ref* ⇒ *bool*

record *state* =
  globals :: *heap*
  p :: *ref*
  levellist :: *ref list*
  ... *local variables/parameters* ...

Fig. 2. Node struct and program state

needs pointers to its children (fields *left* and *right*) and its representative (*rep*), used during the normalization algorithm. The *next* pointer links together all nodes of the same (variable) level in the bdd. This is implemented in procedure *Levellist* as marking algorithm using the *mark* field.

4.2 BDD Model

We follow the approach in [8] to abstract the pointer structure in the heap to HOL datatypes. For the formalization of BDDs we work on two levels of abstraction, the decision tree (BDT) and the graph structure (DAG). On the higher level, we describe the underlying decision tree with the datatype *bdt*:

datatype *bdt* = *Zero* | *One* | *Bdt-Node bdt nat bdt*

A *bdt* is modeled by the constructors *Zero* and *One*, which represent the terminal values *False* and *True*, and by the constructor *Bdt-Node* representing a nonterminal node with two sub-BDTs and the current decision variable.

When looking at this datatype, it becomes clear that we cannot express the concept of sharing by using this content-based definition. Therefore, we introduce another formalization level in order to describe the graph structure of the BDD based on references. For the representation of a BDD in the heap we use datatype *dag*, which is a directed acyclic graph of binary degree:

datatype *datatype dag* = *Tip* | *Node dag ref dag*

A DAG in the heap is either constant *Tip*, which is equal to the Null pointer, or a node consisting of a reference for the root node and two sub-DAGs. This representation allows us to express sharing by equal references in the nodes. Moreover it is convenient to write recursive predicates and functions on a datatype. For example *set-of* yields the references stored in the DAG:

set-of:: *dag* ⇒ *ref set*
set-of Tip = {}
set-of (*Node lt r rt*) = {*r*} ∪ *set-of lt* ∪ *set-of rt*

To actually abstract the pointer structure in the heap to the datatype *dag* we introduce the predicate *Dag*. It constructs a DAG from the initial pointer and the mappings for left and right children:

Dag:: *ref* ⇒ (*ref* ⇒ *ref*) ⇒ (*ref* ⇒ *ref*) ⇒ *dag* ⇒ *bool*
Dag p l r Tip = (*p*=*Null*)
Dag p l r (*Node lt a rt*) = (*p*=a ∧ *p*≠*Null* ∧ *Dag* (*l p*) *l r lt* ∧ *Dag* (*r p*) *l r rt*)

This expression is true when starting at pointer *p* and following heaps *l* and *r* we can construct the DAG passed as fourth argument. The heaps *l* and *r* correspond to the fields *left* and *right* in the state-space.

To construct the decision tree out of the DAG we introduce the function *bdt*. It takes a function indicating the variables assigned to each reference as parameter. Usually we use field *var* in the state-space-record for this purpose. The result of *bdt* is an option type. This implies that not every *dag* encodes a *bdt*. The terminal nodes of the decision tree, *Zero* and *One*, are represented by an inner node *Node Tip p Tip* in the DAG, where *var p* = *0* or *var p* = *1*, respectively. So every proper DAG will end up in those inner nodes.

```
bdt :: dag ⇒ (ref ⇒ nat) ⇒ bdt option
bdt Tip var = None
bdt (Node Tip p Tip) var =
    (if var p = 0 then Some Zero else if var p = 1 then Some One else None)
bdt (Node Tip p (Node l p₂ r)) var = None
bdt (Node (Node l p₁ r) p Tip) var = None
bdt (Node (Node l₁ p₁ r₁) p (Node l₂ p₂ r₂)) var =
    (if var p = 0 ∨ var p = 1 then None
     else case bdt (Node l₁ p₁ r₁) var of None ⇒ None
          | Some t₁ ⇒
            case bdt (Node l₂ p₂ r₂) var of None ⇒ None
            | Some t₂ ⇒ Some (Bdt-Node t₁ (var p) t₂))
```

4.3 Properties on BDDs

We now define predicates and functions on our BDD model that we use for the specification and verification of the normalization algorithm.

Eval. Function *eval* on BDTs expects the BDT which shall be evaluated and an environment (a list containing the values for all variables). It traverses the given BDT following the path indicated by the variable values and finally returns the resulting Boolean value. So *eval t* denotes the Boolean function that is represented by the decision tree *t*.

```
eval :: bdt ⇒ bool list ⇒ bool
eval Zero env = False
eval One env = True
eval (Bdt-Node l v r) env = (if env ! v then eval r env else eval l env)
```

Since all functions in HOL are total, indexing the list in *env* ! *v* will yield an legal but indefinite value when the index is out of range.

An interesting concept which arises from this function is eval-equivalence represented by the operator ∼:

$bdt_1 \sim bdt_2 \equiv eval\ bdt_1 = eval\ bdt_2$

Two BDTs are eval-equivalent if they represent the same Boolean function.

Reduced. We call a DAG *reduced* if left and right non-*Tip* children differ for every node contained in it. Note that the order of equations is significant in this definition.

```
reduced :: dag ⇒ bool
reduced Tip = True
reduced (Node Tip p Tip) = True
reduced (Node l p r) = (l ≠ r ∧ reduced l ∧ reduced r)
```

Ordered. The variable ordering of a given BDD is checked by predicate *ordered*. The root node stores the highest variable and the variables decrease on a path down to the leaf. For the variable information, the function takes a mapping of references to their variables (usually field *var*).

$ordered :: dag \Rightarrow (ref \Rightarrow nat) \Rightarrow bool$
$ordered\ Tip\ var = True$
$ordered\ (Node\ (Node\ l_1\ p_1\ r_1)\ p\ (Node\ l_2\ p_2\ r_2))\ var =$
$\quad (var\ p_1 < var\ p \wedge var\ p_2 < var\ p) \wedge$
$\quad (ordered\ (Node\ l_1\ p_1\ r_1)\ var) \wedge (ordered\ (Node\ l_2\ p_2\ r_2)\ var)$
$ordered\ (Node\ Tip\ p\ Tip)\ var = True$

If the DAG properly encodes a decision tree (according to *bdt*), both children of an inner node will either be *Tip*s or again inner nodes. So we do not have to care about the other cases in the definition above.

Shared. Bryant [3] calls two BDDs isomorphic if they represent the same decision tree. If a BDD is shared, then all isomorphic sub-BDDs will be represented by the same root pointer, i.e. the same DAG. This is encapsulated in predicate *isomorphic-dags-eq*, which should be read "*if two dags are isomorphic, then they are equal*":

$isomorphic\text{-}dags\text{-}eq :: dag \Rightarrow dag \Rightarrow (ref \Rightarrow nat) \Rightarrow bool$
$isomorphic\text{-}dags\text{-}eq\ st_1\ st_2\ var \equiv$
$\forall\, bdt_1\ bdt_2.$
$\quad bdt\ st_1\ var = Some\ bdt_1 \wedge bdt\ st_2\ var = Some\ bdt_2 \wedge (bdt_1 = bdt_2) \longrightarrow$
$\quad st_1 = st_2$

i.e. if the decisions trees resulting from st_1 and st_2 are equal, the two DAGs must also be equal. The sub-DAG structure forms a partial order:

$(t < Tip) = False$
$(t < Node\ l\ p\ r) = (t = l \vee t = r \vee t < l \vee t < r)$
$s \leq t \equiv s = t \vee s < t$

In order to express that a DAG is (maximally) *shared*, we argue that all its sub-DAGs respect the *isomorphic-dags-eq* property:

$shared :: dag \Rightarrow (ref \Rightarrow nat) \Rightarrow bool$
$shared\ t\ var \equiv \forall\, st_1\ st_2.\ st_1 \leq t \wedge st_2 \leq t \longrightarrow isomorphic\text{-}dags\text{-}eq\ st_1\ st_2\ var$

5 Normalization

BDD normalization is a central algorithm of [3] and quite complex in specification and verification. By concentrating on this part, we hope to give an impression of the provability and the verification complexity of pointer programs.

5.1 Overview of the Process of Normalization

We call the process of converting an ordered DAG into an ordered, shared and reduced DAG "normalization". It is encapsulated in procedure *Normalize* which calls on its part the sub-procedures *Levellist*, *ShareReduceRepList* and *Repoint*. The implementation follows the procedure called "reduce" in Bryant's paper [3], but imposes some simplifications and the decomposition into sub-procedures to structure the algorithm and the verification (e.g. in Bryants algorithm steps 2 and 3 below are done simultaneously).

From a high-level point of view, one can divide the normalization process into three main stages:

1. Collect the nodes of the argument DAG according to their variable in a two dimensional level-list. At index n the level-list contains all the nodes of the DAG with variable n.
2. Calculate the representative node for each node in the DAG (and store it in the *rep* field of the node) level by level and bottom up. This means that we work on the breadth of the DAG.
3. Repoint the DAG according to the representatives.

We will now examine these three steps in detail:
Stage 1: Procedure *Normalize* first instantiates the level-list with an empty node list for each variable, that can be contained in the argument DAG. The necessary size of the list is given by the variable stored in the root of DAG, since the DAG is *ordered*. Afterwards, the procedure calls *Levellist*, which fills the level-list with the nodes contained in the DAG. After the call of *Levellist* a node with variable i is contained in the level-list at index i (see figure 3). Note that a node in level i not necessarily denotes the depth in the DAG, since variables do not have to appear strictly consecutive on a path through the DAG.

Via the level-list we can easily access all nodes with the same variable. Those are the ones that may have to be shared. In the DAG structure, the nodes containing the same variable can be contained in different sub-DAGs and therefore are far from each other. With the concept of a level-list, these nodes are much easier to be compared and processed. The inconvenience of this process is the complexity of the conversion from the DAG to the two dimensional list, which is visible in the length of the proof.

Stage 2: After obtaining the two dimensional level-list, we traverse each level looking for a representative for each node, which is then stored in the field *rep*. So we do not change the DAG structure at this stage but just store the pointer

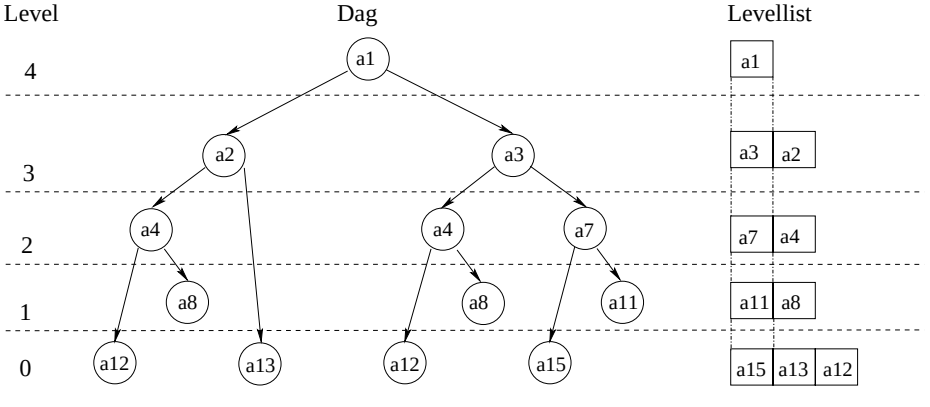
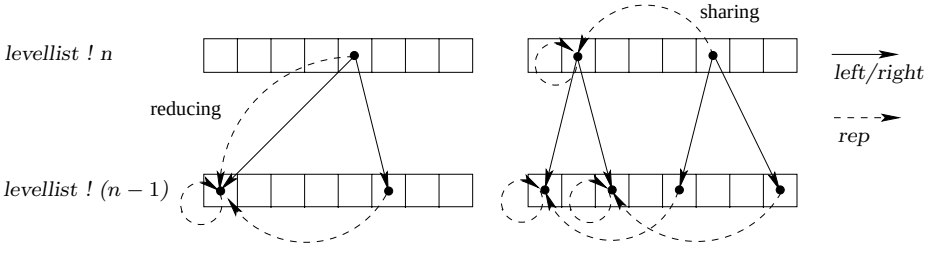


Fig. 3. Illustration of the level-list notion

to the representative node in the field *rep*. Finding the representatives for all nodes in one level of the level-list is realized by procedure *ShareReduceRepList*. It is important to start the normalization with the leaf children (which have the lowest variables), because procedure *ShareReduceRepList* consults the representatives of the children in order to decide if the current node will be shared or reduced. Therefore the children representatives already have to contain the final and correct representatives. Working at a specific level we can assume that the representative DAGs of the lower levels will already be shared and reduced. If both children representatives point to the same node, we reduce, otherwise we search for a node in the current level-list with the same children representatives, which means sharing:



After procedure *Normalize* has traversed all the levels of the level-list, every node contained in the DAG has got a representative by which it will be replaced in the shared and reduced DAG to be constructed. All representative nodes derive from nodes contained in the original DAG. During the process of normalization we never need to construct new nodes.

Stage 3: The only task to complete now, is the "repointerization" of the DAG. We follow the DAG of *rep*-pointers and thereby set the *left* and *right* fields in order to obtain the desired reduced, shared and ordered DAG, which represents our BDD in the heap.

In order to summarize the functionality described above, let us look at the source code of procedure *Normalize*; the auxiliary procedures can be found in the appendix.

procedures *Normalize* $(p \mid p) =$
$\grave{l}evellist := replicate\ (\acute{p} \rightarrow \acute{v}ar + 1)\ Null;$
$\grave{l}evellist := \textbf{\textit{CALL}}\ Levellist\ (\acute{p}, (\neg\ \acute{p} \rightarrow \acute{m}ark)\ ,\ \grave{l}evellist);$
$\acute{n} := 0;$
WHILE $(\acute{n} < length\ \grave{l}evellist)$ ***DO***
  CALL *ShareReduceRepList*$(\grave{l}evellist\ !\ \acute{n});$
  $\acute{n} := \acute{n} + 1$
OD;
$\acute{p} :=$ ***CALL*** *Repoint* $(\acute{p})$

The bar | divides the parameters in value and result parameters. In the case of *Normalize* the input and result parameter is p. The arrow, like in $p \rightarrow var$, mimics

the C-style combined pointer dereferencing and field selection `p->var`. Logically it is equivalent to *var p* in our heap model.

Note that the first code line initializes the the *levellist* array with *Null*-pointers. The level-list is implemented as an array of heap-lists in our programming language, where the array size is fixed by the number of variables. Arrays are represented as HOL-lists.

5.2 Hoare Annotations

Besides the pre- and postcondition around the whole procedure body, we have inserted another Hoare triple around the while loop, starting with ***SPEC***. This inner specification characterizes the important intermediate stages of the algorithm. The precondition captures the point between step 1 and 2, and the postcondition the point between 2 and 3.

In order to be able to distinguish between the different program states we fix state variables σ for the initial procedure state and τ for the program state at the beginning of the inner Hoare triple. This state fixing is part of the assertion syntax: $\{\!|\sigma. \ldots|\!\}$ abbreviates $\{s \mid s{=}\sigma \ldots\}$. State components decorated with the prefix ´ refer to the current state at the position of the assertion. This helps us to speak about different stages of the program state. The logical variables τ and *ll* that are introduced by ***SPEC*** are universally quantified. We will first have a look at the fully annotated procedure before going into detail on its components.

$\forall \sigma\ pret\ prebdt.\ \Gamma \vdash_t$
$\{\!|\sigma.\ Dag\ ´p\ ^1left\ ´right\ pret \wedge ordered\ pret\ ´var \wedge bdt\ pret\ ´var = Some\ prebdt \wedge$
$(\forall no.\ no \in set\text{-}of\ pret \longrightarrow ´mark\ no = ´mark\ ´p)|\!\}$
$^1levellist := replicate\ (´p \rightarrow ´var + 1)\ Null;$
$^1levellist := \textbf{\textit{CALL}}\ Levellist\ (´p, (\neg\ ´p \rightarrow ´mark)\ ,\ ^1levellist);$
$\textbf{\textit{SPEC}}\ (\tau,ll).\ \{\!|\tau.\ Dag\ ^\sigma p\ ^\sigma left\ ^\sigma right\ pret \wedge$
$ordered\ pret\ ^\sigma var \wedge bdt\ pret\ ^\sigma var\ = Some\ prebdt \wedge$
$Levellist\ ^1levellist\ ´next\ ll \wedge$
$wf\text{-}ll\ pret\ ll\ ´var \wedge length\ ^1levellist = (´p \rightarrow ´var) + 1 \wedge$
$wf\text{-}marking\ pret\ ^\sigma mark\ ´mark\ (\neg\ ^\sigma mark\ ^\sigma p) \wedge$
$(\forall pt.\ pt \notin set\text{-}of\ pret \longrightarrow\ ^\sigma next\ pt = ´next\ pt) \wedge$
$^1left = ^\sigma left \wedge ´right = ^\sigma right \wedge ´p = ^\sigma p \wedge ´rep = ^\sigma rep \wedge ´var = ^\sigma var|\!\}$
$´n := 0;$
$\textbf{\textit{WHILE}}\ (´n < length\ ^1levellist)$
$\textbf{\textit{INV}}\ \{\!|Dag\ ^\sigma p\ ^\sigma left\ ^\sigma right\ pret \wedge$
$ordered\ pret\ ^\sigma var \wedge bdt\ pret\ ^\sigma var = Some\ prebdt \wedge$
$Levellist\ ^1levellist\ ´next\ ll \wedge$
$wf\text{-}ll\ pret\ ll\ ´var \wedge length\ ^1levellist = ((´p \rightarrow ´var) + 1) \wedge$
$wf\text{-}marking\ pret\ ^\sigma mark\ ^\tau mark\ (\neg\ ^\sigma mark\ ^\sigma p) \wedge$
$^\tau left = ^\sigma left \wedge ^\tau right = ^\sigma right \wedge ^\tau p = ^\sigma p \wedge ^\tau rep = ^\sigma rep \wedge ^\tau var = ^\sigma var \wedge$
$´n \leq length\ ^\tau levellist \wedge$
$(\forall no \in Nodes\ ´n\ ll.\quad (*\ reduced, ordered\ and\ eval\ equivalent\ *)$
$no \rightarrow ´rep \rightarrow ´var \leq no \rightarrow ´var \wedge$
$(\exists t\ rept.\ Dag\ no\ ^1left\ ´right\ t \wedge$
$Dag\ (´rep\ no)\ (´rep \propto\ ^1left\ )\ (´rep \propto ´right\ )\ rept \wedge$
$reduced\ rept \wedge ordered\ rept\ ´var \wedge$

$$
\begin{array}{l}
\quad (\exists\, nobdt\ repbdt.\ bdt\ t\ `var = Some\ nobdt \wedge \\
\qquad bdt\ rept\ `var = Some\ repbdt \wedge nobdt \sim repbdt) \wedge \\
\quad set\text{-}of\ rept \subseteq `rep\ `\ Nodes\ `n\ ll \wedge \\
\quad (\forall\, no \in set\text{-}of\ rept.\ `rep\ no = no))) \wedge \\
(\forall\, t_1\ t_2. \qquad\qquad\qquad\qquad (*\ shared\ *) \\
\quad \{t_1,t_2\} \subseteq Dags\ (`rep\ `(Nodes\ `n\ ll))\ (`rep \propto `1eft\ )\ (`rep \propto `right) \\
\quad \longrightarrow isomorphic\text{-}dags\text{-}eq\ t_1\ t_2\ `var) \wedge \\
`rep\ `\ Nodes\ `n\ ll \subseteq Nodes\ `n\ ll \wedge \\
(\forall\, pt\ i.\ pt \notin set\text{-}of\ pret \vee (`n \leq i \wedge pt \in set\ (ll\ !\ i) \wedge i < length\ `1evellist) \\
\quad \longrightarrow {}^{\sigma}rep\ pt = `rep\ pt) \wedge \\
`1evellist = {}^{\tau}levellist \wedge `next = {}^{\tau}next \wedge `mark = {}^{\tau}mark \wedge \\
`1eft = {}^{\sigma}left \wedge `right = {}^{\sigma}right \wedge `p = {}^{\sigma}p \wedge `var = {}^{\sigma}var\}
\end{array}
$$

VAR *MEASURE* (*length* `1evellist − `n)
DO
  CALL *ShareReduceRepList*(`1evellist ! `n);
  `n := `n + *1*
OD

$$
\begin{array}{l}
\{(\exists\, rept.\ Dag\ (`rep\ `p)\ (`rep \propto `1eft\ )\ (`rep \propto `right\ )\ rept \wedge \\
\quad reduced\ rept \wedge ordered\ rept\ `var \wedge shared\ rept\ `var \wedge \\
\quad set\text{-}of\ rept \subseteq set\text{-}of\ pret \wedge \\
\quad (\exists\, repbdt.\ bdt\ rept\ `var = Some\ repbdt \wedge prebdt \sim repbdt) \wedge \\
\quad (\forall\, no \in set\text{-}of\ rept.\ (`rep\ no = no))) \wedge \\
ordered\ pret\ {}^{\sigma}var \wedge {}^{\sigma}p \neq Null \wedge \\
(\forall\, no.\ no \in set\text{-}of\ pret \longrightarrow `mark\ no = (\neg\ {}^{\sigma}mark\ {}^{\sigma}p)) \wedge \\
(\forall\, pt.\ pt \notin set\text{-}of\ pret \longrightarrow {}^{\sigma}rep\ pt = `rep\ pt) \wedge \\
`1evellist = {}^{\tau}levellist \wedge `next = {}^{\tau}next \wedge `mark = {}^{\tau}mark \wedge \\
`1eft = {}^{\sigma}left \wedge `right = {}^{\sigma}right \wedge `p = {}^{\sigma}p\}; \\
`p := \textbf{\textit{CALL}}\ Repoint\ (`p) \\
\{(\exists\, postt.\ Dag\ `p\ `1eft\ `right\ postt \wedge \\
\quad reduced\ postt \wedge ordered\ postt\ {}^{\sigma}var \wedge shared\ postt\ {}^{\sigma}var \wedge \\
\quad set\text{-}of\ postt \subseteq set\text{-}of\ pret \wedge \\
\quad (\exists\, postbdt.\ \ bdt\ postt\ {}^{\sigma}var = Some\ postbdt \wedge prebdt \sim postbdt)) \wedge \\
(\forall\, no.\ no \in set\text{-}of\ pret \longrightarrow `mark\ no = (\neg\ {}^{\sigma}mark\ {}^{\sigma}p)) \wedge \\
(\forall\, pt.\ pt \notin set\text{-}of\ pret \longrightarrow {}^{\sigma}rep\ pt = `rep\ pt \wedge {}^{\sigma}left\ pt = `1eft\ pt \wedge \\
\quad {}^{\sigma}right\ pt = `right\ pt \wedge {}^{\sigma}mark\ pt = `mark\ pt \wedge {}^{\sigma}next\ pt = `next\ pt)\}
\end{array}
$$

The Precondition of procedure *Normalize* assumes all the facts that are essential for the call of its sub-procedures: The argument pointer must construct a DAG *pret*, which is ordered, and transformable into the decision tree *prebdt*. Because of *Levellist* being a marking algorithm, all the nodes in this DAG must be identically marked.

The Postcondition. The result of the procedure is a new DAG (*postt*), which is reduced, ordered, and shared. Its nodes are a subset of the nodes of the argument DAG. The decision tree *postbdt* resulting from the new DAG is "eval-equivalent" (operator $\sim$) to the decision tree that we get from the argument DAG, i.e. the Boolean function represented by the normalized BDD is still the same. Besides the marking is inverted in comparison to the beginning of the procedure (which is performed by procedure *Levellist* during the level-list construction). The rest

of the postcondition states that, for nodes which are not contained in the original DAG, the fields that are normally modified by the procedure will remain unchanged. The fact that field *var* does not change, is not captured by this postcondition. We use an additional specification that exploits our split heap model and lists all the global state components that may be modified:

$\forall \sigma.\ \Gamma \vdash \{\sigma\}\ \acute{p} := \textbf{\textit{CALL}}\ \textit{Normalize}\ (\acute{p})$
$\{t.\ t\ \textit{may-only-modify-globals}\ \sigma\ \textit{in}\ [\textit{rep},\textit{mark},\textit{left},\textit{right},\textit{next}]\}$

The verification condition generator makes use of this extra specification [13]. Therefore the regular postcondition only has to mention properties of the global entities that potentially do change. That means, a procedure specification can focus on the relevant portions of the state-space.

The Inner Hoare Triple surrounds the while loop contained in the procedure. Its precondition contains the outer procedure's precondition completed by the results of the call to procedure *Levellist* and some propositions specifying the fields which remained unchanged since the beginning of the procedure. Note that we only have to mention those parts of the state-space here that we refer to in subsequent assertions, i.e. those that are relevant for our current verification task. Procedure *Levellist* adds the following assertions to our precondition:

Levellist $\grave{}$*levellist* $\acute{}$*next ll* The constructed level-list is abstracted to the two dimensional HOL-list *ll* of type *ref list list*. The array $\grave{}$*levellist* contains the initial pointers to the heap lists that link together the nodes of the same level via the $\acute{}$*next* pointers:

$\textit{Levellist levellist next ll} \equiv$
$\textit{map first ll} = \textit{levellist} \wedge (\forall i{<}\textit{length levellist}.\ \textit{List}\ (\textit{levellist}\ !\ i)\ \textit{next}\ (\textit{ll}\ !\ i))$

$\textit{first ps} \equiv \textsf{case}\ \textit{ps}\ \textsf{of}\ [] \Rightarrow \textit{Null} \mid p\ \#\ \textit{rs} \Rightarrow p$

$\textit{List p next}\ [] = (p = \textit{Null})$
$\textit{List p next}\ (a\ \#\ \textit{ps}) = (p = a \wedge p \neq \textit{Null} \wedge \textit{List}\ (\textit{next p})\ \textit{next ps})$

wf-ll pret ll $\acute{}$*var* The level-list is well-formed, i.e. all nodes in the argument DAG are contained in the level-list on their variable position and all nodes in the level-list are contained in the argument DAG:

$\textit{wf-ll pret ll var} \equiv$
$(\forall p.\ p \in \textit{set-of pret} \longrightarrow p \in \textit{set}\ (\textit{ll}\ !\ \textit{var}\ p)) \wedge$
$(\forall k{<}\textit{length ll}.\ \forall p{\in}\textit{set}\ (\textit{ll}\ !\ k).\ p \in \textit{set-of pret} \wedge \textit{var}\ p = k)$

length $\grave{}$*levellist* $= (\acute{p} \rightarrow \acute{}\textit{var}) + 1$ The length of the level-list fits to the variables contained in the DAG.

wf-marking pret $^{\sigma}$*mark* $\acute{}$*mark* $(\neg\ ^{\sigma}\textit{mark}\ ^{\sigma}p)$ All nodes in the DAG are marked contrary to their initial marking:

$\textit{wf-marking pret mark-old mark-new marked} \equiv$
$\textsf{case}\ \textit{pret}\ \textsf{of}\ \textit{Tip} \Rightarrow \textit{mark-new} = \textit{mark-old}$
$\mid \textit{Node lt p rt} \Rightarrow$
$\quad (\forall n.\ n \notin \textit{set-of pret} \longrightarrow \textit{mark-new}\ n = \textit{mark-old}\ n) \wedge$
$\quad (\forall n.\ n \in \textit{set-of pret} \longrightarrow \textit{mark-new}\ n = \textit{marked})$

Now let us think about the inner postcondition. The only action taken after the inner Hoare triple in the source code is the call to procedure *Repoint*. *Repoint* only redirects the original DAG *pret* to the DAG of representatives *rept*, which already has the desired properties: it is reduced, ordered, shared and the resulting decision tree *repbdt* encodes the same Boolean function as the original one. Moreover, since every node in *rept* is a representative the *rep* field of those nodes will point to the node itself. The DAG of representatives *rept* can be obtained out of the original DAG by following the *rep* pointers: *Dag* (*'rep 'p*) (*'rep* $\propto$ *'left*) (*'rep* $\propto$ *'right*) *rept*. We begin with the representative of the root pointer *'rep 'p*, and instead of just following the *left* and *right* pointers we make the additional indirection through *rep*, by the infix operator $\propto$. It is defined as an extension of function composition avoiding to consider representatives of a *Null* pointer:

$rep \propto f \equiv \lambda p.$ *if* $f\ p = Null$ *then* $Null$ *else* $(rep \circ f)\ p$

So in case $left\ p \neq Null$, the expression $(rep \propto left)\ p$ is equivalent to two dereferences: $p \rightarrow left \rightarrow rep$.

In addition we preserve some facts that we already know, like the inversion of the marks, and add the assertion about the parts of the state that are not modified in the loop. Note that we do not modify the DAG structure, since the fields *left* and *right* remain unchanged. Only the *rep* field is modified.

The Loop Invariant starts with the repetition of the facts that we already know from the precondition of the inner Hoare triple. After that the main part of the invariant describes the properties of the processed levels that we have to lift to the current level while proving the invariant and that must suffice to derive the postcondition (of the inner triple) after the loop. Intuitively we have to express that all sub-BDDs stemming from the representative nodes are ordered, reduced and shared and encode the same Boolean function as their original counterparts.

To get hold of the processed nodes and DAGs, we introduce two more predicates, which express that we are processing the original DAG level by level:

- $Nodes\ i\ ll \equiv \bigcup_{k \in \{k \mid k < i\}} set\ (ll\ !\ k)$
  Nodes helps us to speak of all nodes, which are contained in the DAG or level-list up to level i.
- $Dags\ nodes\ left\ right \equiv \{t \mid \exists p.\ Dag\ p\ left\ right\ t \wedge t \neq Tip \wedge set\text{-}of\ t \subseteq nodes\}$
  A DAG is contained in *Dags nodes left right* if its nodes are all contained in *nodes*, if it forms a DAG based on the fields *left* and *right* and if this DAG is no Tip.

For every node *no* that is already processed we know that the representative will not point to a bigger variable; during sharing the variable remains the same and reducing decreases the variable. Starting from a node *no* we can construct the DAG and the decision tree following the original pointers (*t* and *nobdt*) and following the representative pointers (*rept* and *repbdt*). The representative DAG *rept* is ordered and reduced, and the encoded Boolean function is preserved ($nobdt \sim repbdt$). The nodes of *rept* all are representatives of nodes we have

processed so far. This is expressed by *set-of rept* ⊆ *ŕep ‘ Nodes ŕn ll*. Here the infix ‘ is the set image operation. So we can rephrase the set on the right hand side with $\{$*ŕep no* | *no* ∈ *Nodes ŕn ll*$\}$. Moreover the representative of an representative node will point to the node itself. This ensures uniqueness of the representatives.

To properly express the sharing of the representative DAGs, we cannot only refer to a single DAG constructed from a representative node, since we also have to consider sharing between all sub-BDDs. For every two DAGs t_1 and t_2 that we construct from the representative nodes, the sharing property *isomorphic-dags-eq* has to hold.

The remaining parts of the invariant express that the representative nodes are contained in the original nodes, and describe the parts of state that remain unmodified by the loop.

The Loop Variant justifies termination and is specified via a wellfounded relation. In this case a measure function, expressing that the distance of the loop variable to the length of the level-list decreases.

Both Verma et al. [14] as well as Kristic and Matthews [7] encounter some problems regarding termination. They directly map their BDD-algorithms to recursive functions in Coq or Isabelle/HOL respectively. Since the underlying logics only support total functions, they have to come up with a justification for termination upon function definition. The recursive algorithms on BDDs only terminate for proper inputs (e.g. no cycles). Verma et. al. work around this problem by formally defining the recursion on an artificial counter (the variable level). Kristic and Matthews come up with a scheme to simultaneously define the function together with an invariant. By this they are able to handle the nested recursion, that occurs because the global state is an explicit parameter of their functions. Subsequent function application on the left and right sub-DAG results in nested recursion in their approach.

These problems do not occur in our model (e.g. for the auxiliary procedures *Levellist* or *Repoint*), since we do not directly define them as functions in HOL, but just define the piece of syntax making up the procedure body. We can easily restrict the input to well-formed BDDs by the precondition of the Hoare triple, e.g. *Dag ŕp ĺow ĥigh pret* already ensures that there are no cycles in the pointer structure.

6 Conclusion

The verification of partial correctness of the normalization algorithm and its auxiliary procedures sums up to about 10000 lines of Isabelle/Isar formalization and proofs and is based on a master thesis [11]. Adapting the proofs to total correctness is straightforward and only adds a few lines.

We locate the reasons of the complexity mainly in the data structure, which involves a high degree of data sharing and side effects, which results in quite complex invariants, specifications and proofs. We have to keep track of the original BDD the level-list and the representative BDD. As an example our proof

that the property marked as (∗ *shared* ∗) in the invariant is preserved, while we proceed from level n to $n + 1$, required about 1000 lines. We consider two arbitrary Dags up to level $n + 1$ and have to show the *isomorphic-dags-eq* property for them. We make a case distinction, whether both Dags are already in level n, one Dag is already in level n, or none of them is in level n. In the latter case we proceed by inspecting the root nodes to decide whether they where shared or not. Those kind of case distinctions for various properties add up to the large proofs.

To prove the verification conditions, we used the structured language Isar [9] that allows to focus on and keep track of the various aspects of the proof, so that we can conduct it in a sensible order. Moreover it turned out that the Isar proofs are quite robust with regard to the iterative adaption of the invariant resulting from failed proof attempts. The already established lines of reasoning remained stable, while adding new aspects to, or strengthening parts of the invariant. The relatively large size of the proofs is partly explained by the fact that the declarative Isar proofs are in general more verbose than tactic scripts.

The Hoare logic framework and the split heap model appeared to form a suitable verification environment on top of Isabelle/HOL. The abstraction of pointer structures to HOL datatypes allows us to give reasonable specifications. The split heap model addresses parts of the separation problems that occur when specifying procedures on pointer structures. The overhead of describing the parts of the heap that do not change is kept small. The main effort of the work goes into the problem and not into the framework.

The model we used to describe the properties of the BDD pointer structure can serve as a solid basis for more involved BDD algorithms.

References

1. Richard Bornat. Proving pointer programs in Hoare logic. In J. Oliveira R. Backhouse, editor, *Mathematics of Program Construction*, volume 1837 of *Lect. Notes in Comp. Sci.*, pages 102–126. Springer, 2000.
2. Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conference, 1990. Proceedings. 27th ACM/IEEE*, pages 40–45, june 1990.
3. Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
4. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.
5. Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
6. Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mie Barnett, editors, *Formal Methods and Software Engineering 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004*, volume 3308 of *Lect. Notes in Comp. Sci.* Springer, 2004.

7. Sava Krstic and John Matthews. Verifying BDD algorithms through monadic interpretation. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation: Third International Workshop, VMCAI 2002, Venice, Italy, January 21-22*, volume 2294 of *LNCS*, pages 182–195, 2002.
8. Farhad Mehta and Tobias Nipkow. Proving Pointer Programs in Higher-Order Logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *Lect. Notes in Comp. Sci.*, pages 121–135. Springer, 2003.
9. Tobias Nipkow. Structured Proofs in Isar/HOL. In *TYPES 2002*, volume 2646 of *Lect. Notes in Comp. Sci.* Springer, 2002. http://isabelle.in.tum.de/docs.html.
10. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.
11. Veronika Ortner. Verification of BDD Algorithms. Master's thesis, Technische Universität München, 2004. available from http://www.veronika.langlotz.info/.
12. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.
13. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *Lect. Notes in Art. Int.*, pages 398–414. Springer-Verlag, 2005.
14. Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *Proc. 6th Asian Computing Science Conference (ASIAN'2000), Penang, Malaysia, Nov. 2000*, volume 1961, pages 162–181. Springer, 2000.

A Auxiliary Procedures

Levellist traverses the DAG, puts unmarked nodes to the front of the corresponding level-list slot, and switches their mark. Marking ensures that nodes are only collected once and thus no cycles are introduced in the list.

```
procedures Levellist (p, m, levellist | levellist) =
IF (ṕ ≠ Null) THEN
  IF (ṕ→ḿark ≠ ḿ) THEN
    l̀evellist := CALL Levellist (ṕ→l̀eft, ḿ, l̀evellist );
    l̀evellist := CALL Levellist (ṕ→ŕight, ḿ, l̀evellist );
    ṕ→ńext := l̀evellist ! (ṕ→v́ar);
    l̀evellist ! (ṕ→v́ar) := ṕ;
    ṕ→ḿark := ḿ;
  FI
FI
```

ShareReduceRepList processes one level of the level-list. Non-leaf nodes with the same children representatives are reduced, all the others are shared.

isLeaf-pt p left right ≡ *left p* = *Null* ∧ *right p* = *Null*

procedures *ShareReduceRepList* (*nodeslist*) =
´node := *´nodeslist*;
WHILE (*´node* ≠ *Null*) ***DO***
  IF (¬ *isLeaf-pt ´node ˙low ˙high* ∧ *´node*→*˙low*→*´rep* = *´node*→*˙high*→*´rep*)
  THEN *´node* → *´rep* := *´node* → *˙low* → *´rep* (∗ *reducing* ∗)
  ELSE CALL *ShareRep* (*´nodeslist* , *´node*) (∗ *sharing* ∗)
  FI;
  ´node := *´node*→*´next*
OD

ShareRep shares node *p* by searching its representative in the current *nodeslist*. In case of leafs, the representative is the first element in the list. Otherwise the representative is the first node in the list with the same children representatives. Since *p* itself is in the list we will always find a node.

repNodes-eq p q left right rep ≡
(*rep* ∝ *right*) *p* = (*rep* ∝ *right*) *q* ∧ (*rep* ∝ *left*) *p* = (*rep* ∝ *left*) *q*

procedures *ShareRep* (*nodeslist*, *p*) =
IF (*isLeaf-pt ´p ˙low ˙high*)
THEN *´p* → *´rep* := *´nodeslist*
ELSE
  WHILE (*´nodeslist* ≠ *Null*) ***DO***
    IF (*repNodes-eq ´nodeslist ´p ˙low ˙high ´rep*)
    THEN *´p*→*´rep* := *´nodeslist*; *´nodeslist* := *Null*
    ELSE *´nodeslist* := *´nodeslist*→*´next*
    FI
  OD
FI

Repoint traverses the DAG while re-pointing the nodes to their representatives.

procedures *Repoint* (*p* | *p*) =
IF (*´p* ≠ *Null*) ***THEN***
  ´p := *´p*→*´rep*;
  IF (*´p* ≠ *Null*) ***THEN***
    ´p→*˙left* := ***CALL*** *Repoint* (*´p*→*˙left*);
    ´p→*´right* := ***CALL*** *Repoint* (*´p*→*´right*)
  FI
FI

Extensionality in the Calculus of Constructions

Nicolas Oury

Laboratoire de Recherche en Informatique, UMR 8623 CNRS,
Université Paris–Sud, Orsay, France

Abstract. This paper presents a method to translate a proof in an extensional version of the Calculus of Constructions into a proof in the Calculus of Inductive Constructions extended with a few axioms. We use a specific equality in order to translate the extensional conversion relation into an intensional system.

In logical systems based on type theory, terms — and formulas — are identified modulo an equivalence relation which usually includes β–convertibility. Two equivalent terms or formulas are considered as exactly the same. There is also a propositional predicate for equality, such that one can state and possibly prove that two terms are equal. Two equivalent terms are obviously provably equal, the converse is not true in general. Actually, there is a distinction between so-called intensional and extensional type theories: in intensional type theories, two terms are equivalent only if they compute to the same value; in extensional type theory, two terms which are provably equal are equivalent. Having a larger class of equivalent terms leads to simpler proofs, the identification of provably equal terms is also an usual mathematical practice. However, because provability is not decidable, the equivalence relation becomes undecidable in extensional type theories and so is type checking. Except for the Nuprl system [3] and — with a weaker notion of conversion — HOL, all proof assistants based on type theories are indeed implementing an intensional type theory. Nevertheless, there is a growing interest in extending the equivalence relation to include more than β–reduction. Deduction modulo is a logical framework where the equivalence relation plays a central role in deduction, the Calculus of Algebraic Constructions extends also the usual computations with general rewrite rules still preserving normalisation and confluence. A natural question is to understand whether considering equal terms as equivalent significantly changes the logical system. The answer for a first-order system is negative, the extensional system is conservative over the intensional one, which means we can prove exactly the same theorems ([4]). However, the problem is more complicated in type theories with dependent types because having more equivalent terms extends the class of typable formula. M. Hofmann gives a semantical proof of conservativity of Extensional Martin–Lof's type theory into Intensional Martin–Lof's type theory extended with a few axioms. In this paper, we analyse the same problem in the framework of the Calculus of Constructions. Our contribution is to give a syntactic proof: it includes an effective process which translates an extensional proof into an intensional one (with additional axioms), this proof can consequently be checked by a

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 278–293, 2005.

computer. In our proof, we use the so-called John Major's equality introduced by C. McBride in order to compare two terms of different types ; it plays a central role in order to overcome the technical difficulty caused by dependent types.

In section 1, we give a precise presentation of the problem. In section 2, we present the extensional system and expose its properties. In section 3, we extend the intensional *Calculus of Inductive Constructions* in order to be able to translate the extensional calculus into it in section 4. The main result is to establish conservativity of this *Extensional Calculus of Constructions* on this extended *Calculus of Inductive Constructions*.

1 Presentation

The *Calculus of Inductive Constructions* (CIC) is a logical system implemented by the *Coq* proof assistant. CIC is a typed λ–calculus with dependent types and inductive definitions. There is an internal notion of convertibility over terms. This convertibility includes β–conversion and ι–conversion — which deals with pattern matching and fixpoints on inductive definitions. We denote this relation $\equiv$.

Convertibility is automatically used during typing. So some calculi are discharged by the system. For example, because $2+2 \equiv 4$, $2+2 = 4 \equiv 4 = 4$. So the obvious proof of $4 = 4$ is also a proof of $2 + 2 = 4$. This results in shorter proofs. In order for type checking to be decidable, we need $M \equiv N$ to be decidable when M and N are well typed. In the CIC, this is decidable. Indeed, the conversion is based on a reduction $\rightarrow$ strongly normalizing and confluent.

In CIC, it is also possible to define a propositional equality (*Leibniz* equality) defined as the smallest reflexive relation. Two terms that are convertible can be proved equal by reflexivity. But some terms, that are not convertible, can also be proved equal. For example, if we define addition on Peano numbers[1]:

```
plus O n = n
plus (S m) n = plus m (S n)
```

$O + n$ and n are convertible, so the proof of $\forall n, 0 + n = n$ is exactly the same as the proof of $\forall n, n = n$. But $n + O$ and n are not convertible. Indeed, `plus` is defined by pattern matching on its first argument. The first argument of $n + O$ is a free variable, so this term is in normal form. The conversion fails. We have to prove $\forall n, n + O = n$ by induction on n.

Moreover, the *Calculus of Inductive Constructions* allows *dependent types*. So values can appear in types. Hence, two types that depend on values which are equal but not convertible are not convertible. For example, assume we have defined $list\ n$, the type of lists of natural numbers of size n. Size of the list appears in the type. Now, we can define an `append` function to concatenate two of these lists. This function has type:

`append` : $\forall n, m : nat.(list\ n) \rightarrow (list\ m) \rightarrow (list\ (n + m))$

[1] For the sake of clarity, we use pseudo syntax here.

We may want to prove some properties of this function. Let us try to prove the property

$$\forall n : nat,\ l : (list\ n), (append\ l\ nil) = l.$$

Actually, we get into a problem. We can not even write this property. This equality is not well-typed. *append l nil* is of type $list\ (n + O)$ whereas l is of type $list\ n$. As shown earlier, these types are not convertible. So, the types are different. Whereas Leibniz equality only links two terms within the same type.

These errors are difficult to understand and even more difficult to solve. For example, the above property is difficult to formalize. Moreover, we lose modularity. Some properties or proofs relies on the implantation of *plus* and not only on its mathematical behaviour. In order to use natural numbers in a proof development, one need to know their implementation.

Usual mathematics identifies equal terms. So, ideally, a proof system should merge *convertibility* and *Leibniz* equality. Such a system — like *Nuprl* [3] — is said to be *extensional*. In such a system the following conversion would hold :

$$X + 0 \equiv X$$

$$0 + X \equiv X$$

This solves the problem described above.

Martin Hofmann has studied this problem in the context of Martin–Löf type theory [5](Section 3.2.5). He has shown this theory to be conservative over the usual theory extended with Streicher's axiom K and *functional extensionality*[2]. These axioms are similar but weaker than those which are used here. His proof is based on a semantical model of the system. Here, we syntactically translate typed terms in the usual system. We give a practical interest of an effective translation in section 5.

In the following, we drop inductive definitions in the Extensional Calculus of Inductive Constructions. We use proved equalities to reintroduce them later — see section 5.

2 Extensional Calculus of Constructions

We base the *Extensional Calculus of Constructions* (hereafter called CC_E) on the *Extended Calculus of Constructions*(ECC)[7]. This is a Pure Type System (PTS) [1] with a hierarchy of cumulative sorts Type_i and an impredicative sort Prop to express logical propositions.

Figure 1 shows typing rules of the *Extended Calculus of Constructions*. Rule (CONV) allows conversion during typing. The conversion rules for this system are those shown in figure 2 extended with reflexivity, symmetry and transitivity. This conversion is β-conversion. Nevertheless, for the sake of the clarity of our translation, we had rather decomposed it into head reductions and some congruence rules. In the following, we denote the empty context with $\emptyset$, product with

[2] This axiom states that two functions pointwisely equal are equal.

$$\text{(ECON)}\ \frac{}{\vdash \emptyset} \qquad \text{(I–CON)}\ \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A}\ s \in \mathcal{S}$$

$$\text{(TYPE)}\ \frac{}{\vdash \mathit{Type}_i : \mathit{Type}_{i+1}} \qquad \text{(PROP)}\ \frac{}{\Gamma \vdash \mathit{Prop} : \mathit{Type}_0}$$

$$\text{(VAR)}\ \frac{\vdash \Gamma}{\Gamma \vdash x : A}\ x : A \in \Gamma \qquad \text{(UNIV)}\ \frac{\Gamma \vdash A : \mathit{Type}_i}{\Gamma \vdash A : \mathit{Type}_{i+1}}$$

$$\text{(PROD)}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s'}{\Gamma \vdash \forall x : A.B : s'}\ (s, s') \in \mathcal{R}$$

$$\text{(APP)}\ \frac{\Gamma \vdash t : \forall x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B\{x \leftarrow u\}}$$

$$\text{(LAM)}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\mathit{fun}\ x : A \Rightarrow t) : \forall x : A.B}$$

$$\text{(CONV)}\ \frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}$$

Fig. 1. Typing rules of the Extended Calculus of Constructions

$\forall x.T$ and abstraction with $\mathit{fun}\ x \Rightarrow M$. A typing judgement is written $\Gamma \vdash t : T$. $\vdash \Gamma$ are judgements for well formed contexts. $\Gamma \vdash t \equiv t'$ are judgements of convertibility.

2.1 Extensionality Rule

We denote $=$ the usual, propositional *Leibniz* equality in *ECC*. CC_E consists in the Extended Calculus of Constructions extended with an *extensionality* rule:

$$\text{(EXT)}\ \frac{\Gamma \vdash_E t : A = B}{\Gamma \vdash_E A \equiv B}$$

In the following, judgements of CC_E are denoted $\vdash_E$.

If an equality is provable in a context, then the terms are convertible.[3] With this rule, we can type the example on size-dependant lists.

We can also prove that two functions equal pointwise are equal. The following derivation states this property and is a good example of the usage of the extensionality rule[4]:

[3] We keep the name *conversion* for this relation because it is used as a conversion in the usual system. We agree that this is much more powerful than an usual conversion.

[4] In this derivation, for the sake of clarity, we focus on types and forget terms.

$$(\beta)\ \frac{\Gamma \vdash (((fun\ x : A \Rightarrow u)v) : t}{\Gamma \vdash ((fun\ x : A \Rightarrow u)v) \equiv u\{x \leftarrow v\}}$$

$$(\text{CAPP})\ \frac{\Gamma \vdash u_1 \equiv u_1' \qquad \Gamma \vdash u_2 \equiv u_2'}{\Gamma \vdash (u_1\ u_2) \equiv (u_1'\ u_2')}$$

$$(\text{CPROD})\ \frac{\Gamma \vdash u_1 \equiv u_1' \qquad \Gamma,\ x : u_1 \vdash u_2 \equiv u_2'}{\Gamma \vdash \forall x : u_1.u_2 \equiv \forall x : u_1'.u_2'}$$

$$(\text{CLAM})\ \frac{\Gamma \vdash u_1 \equiv u_1' \qquad \Gamma,\ x : u_1 \vdash u_2 \equiv u_2'}{\Gamma \vdash (fun\ x : u_1 \Rightarrow u_2) \equiv (fun\ x : u_1' \Rightarrow u_2')}$$

Fig. 2. Conversion rules of the Extended Calculus of Constructions

$$(\text{EqIntro})\ \frac{(\text{CLAM})\ \dfrac{(\text{EXT})\ \dfrac{\dfrac{\Gamma \vdash_E \forall x : A, M = N}{\Gamma, x : A \vdash_E M = N}}{\Gamma, x : A \vdash_E M \equiv N}}{\Gamma \vdash_E fun\ x : A \Rightarrow M \equiv fun\ x : A \Rightarrow N}}{\Gamma \vdash_E fun\ x : A \Rightarrow M = fun\ x : A \Rightarrow N}$$

2.2 Undecidability

The extensional system allows simpler proofs but has some drawbacks. First of all, in this calculus, typing is undecidable. Intuitively, the extensionality rule (EXT) "forgets" a witness of an equality property. This proof that two terms are equal has been used but is not kept in the proof. In order to typecheck the term, the type system has to guess these equalities and proofs.

More precisely, on one hand, it is easy to prove that a type checker may have to decide for any given Γ, M and N, $\Gamma \vdash_E M \equiv N$. Indeed, let extend Γ with a term P of type $(nat \to nat) \to Prop$ and p of type $\forall x : nat \to nat.P\ x$. It is easy to check that $\Gamma' \vdash_E (fun\ y : (P\ M).y)(p\ N)$ is typable if and only if $\Gamma \vdash_E M \equiv N$.

On the other hand, we can encode the *halting problem* in a convertibility decision. Indeed, let now assume we have defined in CC_E a function T taking 3 arguments. $T\ n\ m\ p$ returns 1 if and only if the n^{th} Turing machine does not halt in p steps or less on entry m. This function can easily be written as a fixpoint on p in CC_E. Let also define, a function f such that $f\ p$ constantly equals 1. Using the derivation of previous section, we have that $\vdash_E T\ n\ m \equiv f$ if and only if $\vdash_E \forall p.T\ n\ m\ p \equiv f\ p$. This latter holds if and only if the n^{th} Turing machine does not halt on entry m. So the type system is able to decide an undecidable problem and so is undecidable.

2.3 Infinite Reductions

Moreover, this system is not strongly normalizing. Some terms can be typed and are not normalizing. For example, in a context containing a proof of $True = True \to True$[5], we can encode the whole λ–*calculus*, which is not normalising.

The Coq proof assistant checks proofs by typing. CC_E typing is undecidable. So this system seems not to be a good choice for this proof assistant. Nevertheless, it is useful as a superset of some decidable type systems.

2.4 A Model for Rewriting Extensions of the Calculus of Inductive Constructions

There is another possibility to solve the convertibility problems shown in section 1. CIC could be extended to support *rewriting rules*. Some extensions — like [2] or [9] — allows the user to add *symbols* and rewriting rules to convert these symbols. In such a system, for example, the user can define *symbols* `S`, `O` and `plus` and add the following *rewriting rules* :

```
O + n -> n
n + O -> n
S n + m -> S (n + m)
m + (S m) -> S (n + m)
```

Since more reductions are allowed, $n + O$ and $O + n$ are both convertible to n. This solves the problem described in section 1.

The system checked some criterias on *symbols* and *rules*. This ensures that the system stays strongly normalizing, confluent and consistent.

Nevertheless, these systems have some drawbacks. Logical power of these system with respect to *Calculus of Inductive Constructions* have not been much studied. Moreover, with such a system, it is not always clear whether we have defined the *same* data types as in the initial system. For example, we can look at the following rewriting system:

```
f 0 -> true
f 1 -> false
```

This rewriting system have an infinity of new normal forms — $f\ i, \forall i > 1$ — for *booleans*. This is not incoherent but very counter-intuitive.

Furthermore, it is difficult to *extract* programs from proofs using rewriting steps. Extraction is a process in the Coq proof assistant that translates a constructive proof to a functional program. Assume we want to extract a proof to a functional language. We know how to translate β–reductions and ι–reductions: they have their counter parts in most functional languages. But, some rewrite rules — like *non–linear* rules — can not be translated in a funcional languages.

We suggest a new way to add rewrite rules. First, the user proves or admits some equalities in the *Calculus of Inductive Constructions*. Then, one of the

[5] This property is not provable in CC_E but is consistent with this system. It holds in some *proof irrelevance* model.

criterias of rewriting extensions is used to check that the system is still *decidable*. The system where we add such *proved* rewrite rules is a subsystem of CC_E. So the translation described here can be used to translate it into a slight extension of the Calculus of Inductive Constructions. It allows to check the proof with a certified kernel. This practical interest will be fully discussed in section 5.

3 Translation of CC_E into CIC+

In section 2, we have shown a derivation that is provable in CC_E and does not hold in CIC. So CC_E is obviously non conservative over CIC. We introduce an extension of CIC, that is powerful enough to encode CC_E. We translate this *extensional* system into that extension of the *Calculus of Inductive Constructions* — thereafter called CIC+. CIC, and so CIC+ is based on the typing rules of ECC — see figures 1 and 2 —, extended with inductive definitions, pattern matching and fixpoints.

We keep the $\vdash$ notation for the deduction in CIC+ and $\vdash_E$ for the deduction in CC_E.

Next, we explain the principle of the translation. Then, we introduce the system CIC+.

3.1 Extensionality Rule

The difference between *extensional* and *intensional* systems is the *extensionality rule*. This rule transforms a proved equality into a *conversion*. Such a transformation is impossible in an *intensional* system like CIC or CIC+. So, we can not translate the conversion of the extensional system into the conversion of the intensional system. The best we can hope is to translate the conversion relation of CC_E into a proved equality in CIC+. We want to have:

$$\Gamma \vdash_E M \equiv N \Rightarrow \exists p, \Gamma \vdash p : M = N$$

This creates some difficulties. Indeed, let assume we have in CIC+ a function *subst* that rewrite with a proved equality.[6] The rule:

$$(\mathrm{CONV})\ \frac{\Gamma \vdash_E t : A \qquad \Gamma \vdash_E A \equiv B \qquad \Gamma \vdash_E B : s}{\Gamma \vdash_E t : B}$$

translates to:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash p : A = B \qquad \Gamma \vdash B : s}{\Gamma \vdash \mathit{subst}\ p\ t : B}$$

The translation changes the proof term. The equality step is made explicit. Since CC_E has dependent types, these explicit conversions can also appears in types. Moreover, the translation of a type depends on the derivation. A same type can have a lot of different translations with different explicit conversions.

[6] We will define this function in the next section.

Nevertheless, we manage to keep a key invariant in the translation.

Invariant. The translated term is the same as the initial term where some subterms t has been replaced by $subst\ p\ t$ for some proof p.

We stress here a point important to understand the whole translation process:

Remark 1. As type checking is undecidable, it can seem strange that there exists a translation into CIC+ — where type checking is decidable. Indeed, the process translates a *type tree* in CC_E. This type tree contains the information about equalities used in conversion. So, the translation have not to guess these equalities, which would be undecidable.

Before explaining the translation, we have to choose an equality for the translation of CC_E conversion relation. A same type in CC_E can have multiple translations. Moreover, *extensional* conversion can happen on two terms in two *intensionally* different types. So this equality has to link terms of different types.

3.2 Definition of = in CIC+

In CIC, the usual equality only allows to compare two terms that have the same type. In our case, we need to compare terms that have not, *a priori*, the same type. For example, $append\ l\ nil$ is of type $list\ (n+O)$ whereas l is of type $list\ n$.

$$(\text{JMType})\ \frac{\Gamma \vdash A, B : Type \qquad \Gamma \vdash x : A \qquad \Gamma \vdash y : B}{\Gamma \vdash x_A =_B y : Prop}$$

$$(\text{JMIntro})\ \frac{\Gamma \vdash x : A}{\Gamma \vdash refl\ x : x_A =_A x}$$

$$(\text{JMElim})\ \frac{\Gamma \vdash e : x_A =_B y \qquad \Gamma \vdash p : P\ A\ x \qquad \Gamma \vdash P : \forall A : Type, A \to Type}{\Gamma \vdash elim\ e\ P\ p : P\ B\ y}$$

$$(\text{JMLeibniz})\ \frac{\Gamma \vdash x, y : A}{\Gamma \vdash x_A =_A y \Rightarrow x =_L y}$$

$$(\text{JMLAM})\ \frac{\begin{array}{c}\Gamma \vdash p_1 : u_1 = u_1' \qquad \Gamma, x : u_1 \vdash u_2 : t \\ \Gamma, y : u_1' \vdash u_2' : t' \qquad \Gamma,\ x : u_1,\ y : u_1', p : x = y \vdash p_2 : u_2 = u_2'\end{array}}{\Gamma \vdash jmlam\ p_1\ p_2 : (fun\ x : u_1 \Rightarrow u_2) = (fun\ y : u_1' \Rightarrow u_2')}$$

$$(\text{JMAPP})\ \frac{\Gamma \vdash p_1 : u_1 = u_1' \qquad \Gamma \vdash p_2 : u_2 = u_2'}{\Gamma \vdash jmapp\ p_1\ p_2 : (u_1\ u_2) = (u_1'\ u_2')}$$

Fig. 3. Essential properties of = in CIC+

$$(\text{JMSET})\ \frac{\Gamma \vdash x_A =_B y}{\Gamma \vdash A =_L B}$$

$$(\text{JMSUBST1})\ \frac{\Gamma \vdash p : A = B \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathit{subst}\ p\ t : B}$$

$$(\text{JMSUBST2})\ \frac{\Gamma \vdash e : A = B \qquad \Gamma \vdash m : A}{\Gamma \vdash \mathit{subst}\ e\ m\ _B =_A m}$$

$$(\text{JMPROD})\ \frac{\begin{array}{c}\Gamma \vdash p_1 : u_1 = u_1' \qquad \Gamma, x : u_1 \vdash u_2 : s \\ \Gamma, y : u_1' \vdash u_2' : s \qquad \Gamma,\ x : u_1,\ y : u_1', p : x = y \vdash p_2 : u_2 = u_2'\end{array}}{\Gamma \vdash \mathit{jmprod}\ p_1\ p_2 : (\forall\ x : u_1.u_2) = (\forall\ y : u_1'.u_2')}$$

Fig. 4. Derivable properties of = in CIC+

Conversion of CC_E links these terms. Hence, in order to translate this relation into a *proved equality*, we have to extend equality to link these terms.

In [8], Conor Mc Bride introduced an equality that allows to compare terms in different types. Nevertheless, two terms can only be equal if they are in the same type. He has called this equality *John Major's equality*. It can be defined in CIC. Nevertheless, in CIC+, we need a slightly stronger equality. Figure 3 states the essential properties of this equality. All these properties are provable in CC_E for Leibniz equality and so need to be true in CIC+.

Definition. CIC+ consists of CIC extended with the equality axiomatised in figure 3.

Figure 4 states some properties — that are provable in CIC+ — that are used in the translation. This defines *subst*, a function that rewrites a term with an equality. One can define this function in CIC+ by using the primitive operation *elim*.

In section 5, we discuss the power of this equality compared to the John Major's equality that can be defined in the CIC.

As we often use this equality we write it =. When there is no ambiguity, we omit the type of the arguments. We write the usual *Leibniz* equality, $=_L$.

4 Proof of the Translation

4.1 Main Difficulty

We first explain the main problem with the translation of CC_E into CIC+. Let assume we inductively translate a derivation of CC_E. For example, one can look at the rule for application :

$$(\text{APP})\ \frac{\Gamma \vdash_E f : \forall x : A, B \qquad \Gamma \vdash_E t : A}{\Gamma \vdash_E f\ t : B}$$

By induction, we have, on one hand, a translation $\Gamma' \vdash f' : \forall x : A', B'$ and, on the other hand, a translation $\Gamma'' \vdash t' : A''$. We have, *a priori*, no link between, respectively, Γ' and Γ'', and A' and A''. Indeed, some explicit conversion — some *subst* p — may have been added at different positions during the translation. In order to use the rule for application in CCI+ and conclude the induction, we have to link them.

In order to solve this problem, we show that all translations of a same term are equal in CIC+. To have this result we have to introduce a new equivalence, linking all terms that are the same *modulo* some explicit conversion steps. This relation formalizes the key invariant of the translation.

4.2 Rewrited Terms Equivalence

Definition 1. *We define the equivalence relation* $\sim$ *on terms CIC+ by induction:*

- *for* x *variable:* $\dfrac{}{x \sim x}$
- *for* t_1, t_2 *and* p *terms,* $\dfrac{t_1 \sim t_2}{subst\ p\ t_1 \sim t_2}$ *and* $\dfrac{t_1 \sim t_2}{t_1 \sim subst\ p\ t_2}$
- *for* m_1, m_2, n_1 *and* n_2 *terms:* $\dfrac{m_1 \sim m_2 \qquad n_1 \sim n_2}{(m_1\ n_1) \sim (m_2\ n_2)}$
- *for* A_1, A_2, t_1 *and* t_2 *terms:* $\dfrac{A_1 \sim A_2 \qquad t_1 \sim t_2}{(fun\ x : A_1 \Rightarrow t_1) \sim (fun\ x : A_2 \Rightarrow t_2)}$ *and* $\dfrac{A_1 \sim A_2 \qquad t_1 \sim t_2}{(\forall x : A_1.t_1) \sim (\forall x : A_2.t_2)}$

This relation is canonically extended to contexts.

Remark 2. We stress the fact that this relation is purely syntactic and does not rely on typing.

Remark 3. subst is defined and one can be puzzled by the fact it could be inlined or reduced. The following translation can be done with an opaque parameter *subst′* — that would have the same properties as *subst* but can not be reduced. We can set this parameter to *subst* at the end of the translation.

We now prove that two typable equivalent terms are equal.

Lemma 1. *Let* t_1 *and* t_2 *be two terms. If* $t_1 \sim t_2$, $\Gamma \vdash t_1 : T_1$ *and* $\Gamma \vdash t_2 : T_2$, *then there exists* p *such that* $\Gamma \vdash p : t_1 = t_2$.

To prove this lemma, we need to introduce a larger relation. For all set E of couple of variables, we denote $\sim_E$ the relation $\sim$ extended by $x \sim y$ for every $(x, y) \in E$. In fact, we prove, by induction on the formation rules of $\sim_E$, that, for all E, for all Γ, if $\forall (x, y) \in E, \Gamma \vdash \_ : x = y$, $t_1 \sim_E t_2$, $\Gamma \vdash t_1 : \_$ and $\Gamma \vdash t_2 : \_$ then $\Gamma \vdash \_ : t_1 = t_2$.

- For $x \sim_E y$ the result is an hypothesis and for $x \sim_E x$ the result is an application of reflexivity.

- For $t_1 \sim_E \mathit{subst}\ p\ t_2$, we have by induction hypothesis $\Gamma \vdash \_ : t_1 = t_2$ and we conclude by use of property $(JMSubst2)$.
- For $(m_1 n_1) \sim_E (m_2 n_2)$, we have $\Gamma \vdash m_1 = m_2$ and $\Gamma \vdash n_1 = n_2$ by induction hypothesis, we conclude by $(JMProd)$.
- For $\forall x : T_1.m_1 \sim_E \forall x : T_2.m_2$, we use α–conversion and prove $\forall x : T_1.m_1 \sim_E \forall y : T_2.m_2 x \leftarrow y$. Let E' be $E \cup \{(x, y)\}$. By induction hypothesis, we have $\Gamma \vdash \_ : T_1 = T_2$ and $\Gamma, x : T_1, y : T_2, \_ : x = y \vdash \_ : m_1 = m_2$. We conclude by using the contextual rule of equality for product.
- The proof of λ–rule case is similar to the one of product.

We prove the original lemma by choosing $E = \emptyset$.

Lemma 2. *If $t_1 \sim t'_1$ and $t_2 \sim t'_2$ then $t_1\{x \leftarrow t_2\} \sim t'_1\{x \leftarrow t'_2\}$.*

By induction on the formation rules of $t_1 \sim t'_1$. For $y \sim y$, then we conclude because $y \sim y$ and $t_2 \sim t'_2$. For $\mathit{subst}\ p\ t \sim t'$, we have $t\{x \leftarrow t_2\} \sim t'\{x \leftarrow t'_2\}$.So we have $\mathit{subst}\ p\{x \leftarrow t_2\}\ t\{x \leftarrow t_2\} \sim t'\{x \leftarrow t'_2\}$. The other cases are simple applications of contextual rules.

4.3 Set of Terms with Explicit Conversions

We have now a tool to link judgements in CIC+ that are the same *modulo* some explicit conversions. We now translate a judgement in CC_E by a set of terms in CIC+ that are $\sim$–equivalent. Such a translation ensures — see the previous subsection — that all translations of a judgement can be proved to be equal in CIC+.

We have to translate three kinds of — mutually recursive — judgements: context formation, typing and conversion. We have to show that these interpretations sets are never empty — whatever choices have already been made in the translation process.

Definition 2. *For any $\vdash_E \Gamma$ we define a set $[\![\vdash_E \Gamma]\!]$ of judgements valid for CIC such that $\vdash \overline{\Gamma} \in [\![\vdash_E \Gamma]\!]$ if and only if $\overline{\Gamma} \sim \Gamma$ and Γ and $\overline{\Gamma}$ bind the same variable and each binded variable have the same kind.*

For any $\Gamma \vdash_E t : T$ we define a set $[\![\Gamma \vdash_E t : T]\!]$ of judgements valid for CIC such that $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_E t : T]\!]$ if and only if $\vdash \overline{\Gamma} \in [\![\vdash_E \Gamma]\!]$, $\overline{t} \sim t$ and $\overline{T} \sim T$.

Lemma 3. *We can always choose types $\overline{T}$ that have the same head constructor as T.*

Proof of the lemma Assume we have $\overline{\Gamma} \vdash \overline{t} : \overline{T}$. By definition of $\sim$, for any terms $T \sim \overline{T}$, $\overline{T}$ is shaped $\mathit{subst}\ p_1\ \ldots\ \mathit{subst}\ p_n \overline{T'}$ with $\overline{T'}$ having the same head constructor than T. Any subterm of a typable term is typable. So $\overline{T'}$ is typable. Moreover, from lemma 1, it is equal to $\overline{T}$. So there exists p of type $T = \overline{T'}$ such that $\overline{\Gamma} \vdash \mathit{subst}\ p\ \overline{t} : \overline{T'}$.

We now prove our translation process.

Theorem 1. *The following properties are valid:*

1. *If $\Gamma \vdash_E t : T$ then for any $\vdash \overline{\Gamma}$ in $[\![\vdash_E \Gamma]\!]$ there exists $\overline{t}$ and $\overline{T}$ such that $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_E t : T]\!]$.*

2. *If* $\vdash_E \Gamma$ *then there exists* $\vdash \overline{\Gamma} \in [\![\vdash_E \Gamma]\!]$.
3. *If* $\Gamma \vdash_E t_1 \equiv t_2$ *there exists* $\overline{\Gamma} \vdash \overline{t} : \overline{t_1} = \overline{t_2} \in [\![\Gamma \vdash t : t_1 = t_2]\!]$

Corollary 1. *For any* $\overline{\Gamma} \vdash \overline{T} : s$ *in* $[\![\Gamma \vdash_E T : s]\!]$, *and forall* t *such that* $\Gamma \vdash_E t : T$, *there exists* $\overline{t}$ *such that* $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_E t : T]\!]$. *In particular, if* $\Gamma \vdash T : s$ *there exists* $\overline{t}$ *such that* $\Gamma \vdash \overline{t} : T \in [\![\Gamma \vdash_E t : T]\!]$.

Proof of the corollary. By property 1 of the theorem, we have a translation. By lemma 2, we can choose $\overline{T}$.

This corollary induces the *conservativity* of CC_E over CIC+.

Proof of the theorem. We proceed by induction on the typing rules of CC_E.

1. (ECON) $\dfrac{}{\vdash_E \emptyset}$ become $\dfrac{}{\vdash \emptyset}$
2. (I-CON) $\dfrac{\Gamma \vdash_E A : s}{\Gamma, x : A \vdash_E}$. By induction hypothesis, there exists $\overline{\Gamma} \vdash \overline{A} : s$ in $[\![\Gamma \vdash_E A : s]\!]$. By the context formation rule we get $\vdash \Gamma, x : \overline{A}$. Moreover $\overline{A}$ and A have the same kind.
3. (TYPE) and (PROP) These rules translate to the same rules in CIC+.
4. (UNIV) $\dfrac{\Gamma \vdash A : Type_i}{\Gamma \vdash A : Type_{i+1}}$ Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis, we have $\overline{\Gamma} \vdash \overline{A} : Type_i \in [\![\Gamma \vdash A : Type_i]\!]$. We conclude by using universe cumulativity in CIC.
5. (VAR) $\dfrac{\vdash_E \Gamma}{\Gamma \vdash_E x : A}$. Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. Then we get by application of the same rule $\overline{\Gamma} \vdash x : \overline{A} \in [\![\Gamma \vdash_E x : A]\!]$.
6. (PROD) $\dfrac{\Gamma \vdash_E A : s \qquad \Gamma, x : A \vdash_E B : s'}{\Gamma \vdash_E \forall x : A.B : s'}$
   Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis, there exists $\overline{A}$ such that $\overline{\Gamma} \vdash \overline{A} : s \in [\![\Gamma \vdash_E A : s]\!]$.
   Moreover, as $\vdash \overline{\Gamma}, x : \overline{A} \in [\![\vdash_E \Gamma, x : A]\!]$, there exists $\overline{B}$ such that $\overline{\Gamma}, x : \overline{A} \vdash \overline{B} : s' \in [\![\Gamma, x : A \vdash_E B : s']\!]$. As $(s, s') \in \mathcal{S}$, we conclude $\overline{\Gamma} \vdash \forall x : \overline{A}.\overline{B} : s' \in [\![\Gamma \vdash_E \forall x : A.B : s']\!]$.
7. (LAM) $\dfrac{\Gamma, x : A \vdash_E t : B}{\Gamma \vdash_E (fun\ x : A \Rightarrow t) : \forall x : A.B}$
   Let $\vdash \overline{\Gamma}, x : \overline{A}$ be in $[\![\vdash_E \Gamma, x : A]\!]$. By induction hypothesis, there exists $\overline{t}$ and $\overline{B}$ such that $\overline{\Gamma}, x : \overline{A} \vdash \overline{t} : \overline{B} \in [\![\Gamma, x : A \vdash_E t : B]\!]$. Then, with the λ–intro rule, we have : $\overline{\Gamma} \vdash (fun\ x : \overline{A} \Rightarrow \overline{t}) : \forall x : \overline{A}.\overline{B} \in [\![\Gamma \vdash_E (fun\ x : A \Rightarrow t) : \forall x : A.B]\!]$
8. (APP) $\dfrac{\Gamma \vdash_E t : \forall x : A.B \qquad \Gamma \vdash_E u : A}{\Gamma \vdash_E (t\ u) : B\{x \leftarrow u\}}$

Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis and *lemma 3*, there exists $\overline{t}$, $\overline{A}$ and $\overline{B}$ such that $\overline{\Gamma} \vdash \overline{t} : \forall x : \overline{A}.\overline{B} \in [\![\Gamma \vdash_E t : \forall x : A.B]\!]$. Moreover, there exist u' and A' such that $\overline{\Gamma} \vdash u' : A' \in [\![\Gamma \vdash_E u : A]\!]$ As we have $\overline{\Gamma} \vdash \overline{A} = A'$, there exists $\overline{u}$ such that $\overline{\Gamma} \vdash \overline{u} : \overline{A} \in [\![\Gamma \vdash_E u : A]\!]$.
So, with the application rule, we conclude:

$$\overline{\Gamma} \vdash (\overline{t}\ \overline{u}) : \overline{B}[x \leftarrow \overline{u}] \in [\![\Gamma \vdash_E (t\ u) : B[x \leftarrow u]]\!]$$

9. (CONV) $\dfrac{\Gamma \vdash_E A \equiv B \qquad \Gamma \vdash_E t : A \qquad \Gamma \vdash_E B : s}{\Gamma \vdash_E t : B}$

Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis and *lemma 3*, there exist p, $\overline{A}$ and $\overline{B}$, such that $\overline{\Gamma} \vdash p : \overline{A} = \overline{B} \in [\![\Gamma \vdash_E A \equiv B]\!]$
Moreover, there exist, by induction hypothesis, $\overline{t'}$ and $\overline{A'}$ such that $\overline{\Gamma} \vdash \overline{t'} : \overline{A'} \in [\![\Gamma \vdash_E t : A]\!]$.
Moreover, $\overline{\Gamma} \vdash \_ : \overline{A} = \overline{A'}$. We conclude, there exists $\overline{t}$ such that $\overline{\Gamma} \vdash \overline{t} : \overline{A} \in [\![\Gamma \vdash_E t : A]\!]$. So, $\overline{\Gamma} \vdash (\mathit{subst}\ p\ \overline{t}) : \overline{B} \in [\![\Gamma \vdash_E t : B]\!]$.

10. (EXT) $\dfrac{\Gamma \vdash_E t : A = B}{\Gamma \vdash_E A \equiv B}$

Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis, there exist $\overline{t}$, $\overline{A}$ and $\overline{B}$ such that $\overline{\Gamma} \vdash \overline{t} : \overline{A} = \overline{B} \in [\![\Gamma \vdash_E A = B]\!]$. By definition, this judgement is also in $[\![\Gamma \vdash_E A \equiv B]\!]$.

11. (CAPP) $\dfrac{\Gamma \vdash_E u_1 \equiv u_1' \qquad \Gamma \vdash_E u_2 \equiv u_2'}{\Gamma \vdash_E (u_1\ u_2) \equiv (u_1'\ u_2')}$

We conclude by induction hypothesis and congruence rule (JMAPP).

12. (CPROD) $\dfrac{\Gamma \vdash_E u_1 \equiv u_1' \qquad \Gamma,\ x : u_1 \vdash_E u_2 \equiv u_2'}{\Gamma \vdash_E \forall x : u_1.u_2 \equiv \forall x : u_1'.u_2'}$

This rule is, in an extensional system, equivalent to:

$$\frac{\Gamma \vdash_E u_1 \equiv u_1' \qquad \Gamma,\ x : u_1,\ y : u_1', p : x = y \vdash_E u_2 \equiv u_2'\{x \leftarrow y\}}{\Gamma \vdash_E \forall x : u_1.u_2 \equiv \forall x : u_1'.u_2'}$$

We have, by induction hypothesis $\overline{\Gamma} \vdash \_ : \overline{u_1} = \overline{u_1'}$ and — as $x = y$ is typable in $\overline{\Gamma},\ x : \overline{u_1},\ y : \overline{u_1'}$ — $\overline{\Gamma},\ x : \overline{u_1},\ y : \overline{u_1'},\ \overline{p} : x = y \vdash \_ : \overline{u_2} = \overline{u_2'}\{x \leftarrow y\}$. We conclude by application of congruence rule (JMPROD).

13. (CLAM) The contextual rule for functions is similar to the rule for products.
14. (β)

$$\frac{\Gamma \vdash_E ((\mathit{fun}\ x : A \Rightarrow u)v) : \_}{\Gamma \vdash_E ((\mathit{fun}\ x : A \Rightarrow u)v) \equiv u\{x \leftarrow v\}}$$

From *lemma 3*, there exist $\overline{\Gamma}$, $\overline{A}$, $\overline{u}$, $\overline{B}$, $\overline{A'}$ et $\overline{v'}$ such that: $\overline{\Gamma} \vdash (\mathit{fun}\ x : \overline{A} \Rightarrow \overline{u}) : \forall x : \overline{A}.\overline{B}$ and $\Gamma \vdash \overline{v} : \overline{A'}$
As $\overline{\Gamma} \vdash \overline{A} = \overline{A'}$, there exist $\overline{v}$ such that: $\overline{\Gamma} \vdash ((\mathit{fun}\ x : \overline{A} \Rightarrow \overline{u})\overline{v}) : \overline{B} \in [\![\Gamma \vdash_E ((\mathit{fun}\ x : A \Rightarrow u)v) : \_]\!]$
By β–reduction, $\overline{\Gamma} \vdash \_ : ((\mathit{fun}\ x : \overline{A} \Rightarrow \overline{u})\overline{v}) = \overline{u}\{x \leftarrow \overline{v}\}$

5 Discussions and Conclusion

5.1 Comparison Between = in CIC+ and John Major's Equality Defined in CIC

In this translation, we have extended CIC to support a stronger equality. This equality links terms of different types and so it is similar to John Major's equality. One can define this latter in CIC but some of the properties of figure 3 do not hold.

- (JMType), (JMIntro) and (JMElim) holds.
- (JMLeibniz) does not hold in CIC. We need this axiom in order to prove (JMSubst2). It is equivalent to Streicher's axiom K. [6]
- (JMLam) does not hold in CIC. This is often called *functional extensionality* since it states that two functions pointwise equal are equal.
- (JMApp) can not be proved in CIC.

So CIC+ is CIC extended with three axioms.

1. Streicher's axiom K
2. Functional extensionality
3. (JMAPP), the equality of the results of two equal applications

In [5], only the two former are needed.

We now give a justification of these axioms by giving their interpretation in a *proof irrelevance* model of CIC in *set theory*.

- Axiom K states that the only proof of $t = t$ is $refl\ t$. It is clear in a *proof irrelevance* model, where two proofs of a proposition in Prop are always equal.
- Functional extensionality states that two functions that have equal results are equal. This is the definition of equality for functions in set theory.
- (JMAPP) states two equal functions applied to two equal arguments returns equal results. This holds for equality in set–theory.

5.2 Practical Interest of This Translation

This translation has some practical interests.

CC_E can be used as a superset of the rewriting extensions to CIC. Indeed, let assume the user has extended CIC with some new symbols and new rewrite rules linking these symbols. If the user can give an interpretation of these symbols in CIC and prove the equalities corresponding to these rewrite rules, then CC_E is a superset of CIC extended with these symbols and rewrite rules. So the translation process can translate a proof in this extended system into a proof in CIC+ that uses the equalities the user has proved. This allows a certified kernel to check these translated proofs. The extended system can construct a proof term that uses rewrite rules. This proof term can then be translated to be checked in the old, certified kernel.

Moreover, this translation gives a way to implement program extraction from proof in these extensions of CIC. Indeed, programs can be extracted from proofs

in CIC. Since axioms added in CIC+ are valid for *observational equivalence* of programs, the same process can be used to extract program in CIC+. Moreover, the shape of the translated terms guarantees that the extraction is safe in CC_E. Indeed, terms in CC_E can be translated into CIC+ by adding some explicit conversions *subst*. Since t and *subst* p t have the same extracted program, we can use the extraction process of CIC to extract proof in CC_E. Since CC_E can be used as a superset of rewriting extensions of CIC, this gives a safe way to extract programs from proofs in one of these extensions.

5.3 Generalizations

In the system we studied, some simplifications have been made. We work out these details here.

Universes Cumulativity. In the *Calculus of Inductive Constructions*, there is a hierarchy of universes $Type_0 \subset Type_1 \subset \ldots$. This is achieved by converting from $Type_i$ to $Type_j$ when $j > i$. But this means that conversion is not an equivalence relation anymore. This is a reduction order. As it would prevent us to encode conversion into equality, we choosed another rule: $\dfrac{\Gamma \vdash T : Type_i}{\Gamma \vdash T : Type_{i+1}}$

This was the original rule of the *Calculus of Constructions*. Luo explained in [7] that this is not equivalent to adding cumulativity in reduction. For example:

$$X : Type_0 \to Type_0 \nvdash X : Type_0 \to Type_1$$

Nevertheless, this restriction is not a restriction in our case. We still have:

$$X : Type_0 \to Type_0 \vdash (\mathit{fun}\ x \Rightarrow X\ x) : Type_0 \to Type_1$$

We also have by extensionality:

$$(\mathit{fun}\ x \Rightarrow X\ x) = X$$

So we can replace X everywhere. When a property contains $(\mathit{fun}\ x \Rightarrow X\ x)$, we can replace it by X with the extensionality rule. With this restriction we get a great technical simplification with no restriction on expressiveness.

Inductive Definitions. In this paper, we have decided, for the sake of simplicity, not to include *inductive types* in CC_E. In fact, since we can include equalities in convertibility, there is no need to include inductives in CC_E.

They can be introduced in CC_E. We can introduce axioms for constructors, *elimination*, *recursion* and *induction*. Then, we introduce axiomatised equalities for these symbols. Since this system is *extensional*, these equalities are automatically included in the conversion in CC_E. This axiomatisation will be used in the translation but is provable using inductive types in CIC+. So we can translate terms of CC_E extended with inductive types.

5.4 Conclusion

In this paper, we have shown a translation of an *Extensional Calculus of Constructions* into the usual *Calculus of Inductive Constructions* extended with Streicher's axiom K and contextual rules that allow John Major's equality to be a congruence.

This introduces a new approach to extend the Calculus of Inductive Constructions with rewrite rules. As soon as the set of rewriting rules is provable in CIC+, we can safely include these rewrite rules in CIC+. This leads to a method to safely introduce rewriting in the Coq proof assistant. First we provide a model and prove some equalities. Then we use a criteria to ensure that the equalities induces a normalising and decidable system. This process still allows to extract programs from proofs in an usual functional language.

References

1. Henk Barendregt. Typed lambda calculi. In Abramsky et al., editor, *Handbook of Logic in Computer Science*. Oxford Univ. Press, 1993.
2. F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In Paliath Narendran and Michael Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631, Trento, Italy, July 1999.
3. Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
4. Gilles Dowek. *La part du calcul*. Thèse d'habilitation, Université Paris 7, 1999.
5. M. Hofmann. *Extensional concepts in intensional type theory*. Phd thesis, Edinburgh university, 1995.
6. Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *9th Symposium on Logic in Computer Science (LICS)*, Paris, 1994.
7. Zhaohui Luo. ECC an Extended Calculus of Constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, California, 1989.
8. Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.
9. Daria Walukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. In *Proceedings of the Workshop on Logical Frameworks and Meta-languages, Santa Barbara, California*, 2000. Part of the LICS'2000.

A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic

Tom Ridge[1] and James Margetson

[1]LFCS, Informatics, Edinburgh University, Scotland, UK

Abstract. We present a system of first order logic, together with soundness and completeness proofs wrt. standard first order semantics. Proofs are mechanised in Isabelle/HOL. Our definitions are computable, allowing us to derive an algorithm to test for first order validity. This algorithm may be executed in Isabelle/HOL using the rewrite engine. Alternatively the algorithm has been ported to OCaML.

1 Introduction

In this work we mechanise a system of first order logic, and show soundness and completeness for this system. We also derive an algorithm which tests a sequent s for first order validity: s is true in all models iff our algorithm terminates with the answer *True*. All results are mechanised in Isabelle/HOL, and the theorem prover can be executed inside Isabelle/HOL using the rewrite engine. Alternatively, the definitions have been ported to OCaML to give a directly executable theorem prover.

This work is interesting for a number of reasons. Soundness is a prerequisite for a logical system. Completeness of a logical system means that any sequent true in all models is provable in the system. This signals a step change in confidence in the system: when attempting a proof of a true statement, we have gone from knowing that we will never err, to knowing that we will eventually succeed. Soundness and completeness for first order logic are the first significant results in metamathematics, so that the mathematical content of this work is interesting.

Our main contribution is to take this process one step further, and provide a mechanically verified algorithm that will actually prove every valid sequent. This is the first mechanically verified, sound and complete theorem prover for FOL. Others have presented mechanisations of completeness proofs for propositional logic, and occasionally predicate logic, but none have aimed to make the definitions executable. Completeness of a theorem prover is useful from a user's point of view: one wants to know that the failure of a theorem prover to prove a sequent arises from the unprovability of the sequent, and not from a deficiency in the theorem prover.

Our work is also interesting because of the range of possible applications.

> The mechanization of metamathematics itself has important implications for automated reasoning since metatheorems can be applied as labor-saving devices to simplify proof construction.[Sha94]

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 294–309, 2005.

For instance, reflection [Har95] is a mechanism whereby, having verified a piece of code correct, one can incorporate it in a trusted way in the kernel of a theorem prover or proof checker. Since we have verified a theorem prover for first order logic, we could conceivably incorporate this code into the kernel of a theorem prover or proof checker. The advantage of reflection is that we can safely extend our systems in non-trivial ways.

Proofs involving the semantics of logical systems are subtle, and can be hard to construct correctly, and to understand correctly, because one tends to bring a significant amount of intuition to the process, which may not be justified. As an example of these problems, free variables have long been felt to be problematic [ZTA$^+$], so much so that some formalisations of first order logic go to great lengths to avoid them all together [Qui62]. We feel that this work has pedagogic advantages in this area, and have attempted to illustrate this with a completely formal proof of the soundness of the $\forall R/\forall I$ rule[1].

We feel the mechanisation is also a contribution.

- We polish the proofs substantially: we were not afraid to change the definition of the logical system to make the proofs much more pleasant.
- The mechanisation is small, consisting of around 1000 lines of definitions and proofs, which makes comprehending and extending the work hopefully as simple as possible.
- We highlight dependencies between sections of the proof: soundness, for instance, does not require the universe of the models to be infinite.
- For metamathematical reasons, we aim to make the proofs as weak as possible. We remove uses of wellfounded induction in favour of natural number induction, which is the strongest principle we use, save for one application of König's lemma. Consequently the proofs could be carried out in relatively weak systems, certainly much weaker than HOL.

This work is also interesting from an aesthetic standpoint, in that it nicely combines the areas of mathematics, metamathematics, logic, and algorithms. In the following sections, we give *all* the definitions, and outline the main lemmas, of the mechanised proofs.

2 Proof Outline

The rules of our logical system are given in Fig. 1. Terms are simply variables x_i. Theoretically this is no restriction, since the usual first order terms may be simulated. We occasionally use x, y, z to stand for variables. Parameter a is a variable x_i. a is used in preference to emphasise the eigenvariable status: a does not appear free in $\forall x.A, \Gamma$. Atomic predicates are positive atoms $P, Q, \ldots$ and negative atoms $\overline{P}, \overline{Q}, \ldots$. Literals are atomic predicates applied to a tuple of terms, $P(x_{i_1}, \ldots, x_{i_n}), \overline{P}(x_{i_1}, \ldots, x_{i_n}), \ldots$. The intent is that $P(x_{i_1}, \ldots, x_{i_n})$ is true in a model iff $\overline{P}(x_{i_1}, \ldots, x_{i_n})$ is false. Note that there is no relation between

[1] Which is simply the $\forall$ rule here, since we work in a one sided system.

$$\frac{}{\vdash P(x_{i_1},\ldots,x_{i_k}),\Gamma,\overline{P}(x_{i_1},\ldots,x_{i_k}),\Delta}\ Ax \qquad \frac{\vdash \Gamma,P(x_{i_1},\ldots,x_{i_k})}{\vdash P(x_{i_1},\ldots,x_{i_k}),\Gamma}\ NoAx$$

$$\frac{}{\vdash \overline{P}(x_{i_1},\ldots,x_{i_k}),\Gamma,P(x_{i_1},\ldots,x_{i_k}),\Delta}\ \overline{Ax} \qquad \frac{\vdash \Gamma,\overline{P}(x_{i_1},\ldots,x_{i_k})}{\vdash \overline{P}(x_{i_1},\ldots,x_{i_k}),\Gamma}\ \overline{NoAx}$$

$$\frac{\vdash \Gamma,A,B}{\vdash A\vee B,\Gamma}\ \vee \qquad \frac{\vdash \Gamma,A \quad \vdash \Gamma,B}{\vdash A\wedge B,\Gamma}\ \wedge$$

$$\frac{\vdash \Gamma,[a/x]A}{\vdash \forall x.A,\Gamma}\ \forall \qquad \frac{\vdash \Gamma,[x_n/x]A,(\exists x.A)^{n+1}}{\vdash (\exists x.A)^n,\Gamma}\ \exists$$

$NoAx$: $\overline{P}(x_{i_1},\ldots,x_{i_k})$ does not appear in Γ
$\overline{NoAx}$: $P(x_{i_1},\ldots,x_{i_k})$ does not appear in Γ
$\forall$: a does not appear free in $\forall x.A,\Gamma$

Fig. 1. Deterministic Variant of Wainer and Wallen's System [WW92]

an atomic predicate applied to two tuples of different arity. For instance, the value of $P(x,y)$ in a model is independent of the value of $P(x,y,z)$. Formulas $A,B,\ldots$ are inductively defined as the least set containing literals, and closed under applications of $\wedge,\vee,\forall,\exists$. The omission of $\neg,\rightarrow$ is no restriction, since they can be defined as abbreviations as usual. Numbered formulae are pairs of a formula and a number. We say that a formula A is tagged with n, and write A^n, when talking about the numbered formula (A,n). Sequents $\vdash \Gamma$ are lists Γ of numbered formulae. Initially every formula in a sequent is tagged with 0. Any formulae which arise as a result of applying rules $\wedge,\vee,\forall,\exists$ get tagged with 0. Except when the formula is quantified by an $\exists$, the tag is irrelevant and is not displayed.

A derivation in this system is simply a *finite* tree constructed using the rules. If we read these rules backwards, they give an algorithm for taking apart a sequent. The algorithm is deterministic[2] because exactly one rule applies to any given sequent.

We can connect syntax and semantics in the standard way by giving an interpretation of primitive propositions as propositions in a model and extending this to formulas and sequents such that the extension respects standard Tarski semantics. For example, $A\wedge B$ is true in a model iff A is true, and B is true.

Looking at the rules, it is quite easy to convince ourselves that they are *sound* in the sense that, if the premises of the rule are true in all models, with respect to all interpretations of free variables, then so too is the conclusion. For the axioms, this is just the recognition that we are working in a classical metalogic. Actually, soundness for rule $\forall$ is not so obvious, and we give a full proof later.

[2] In rule $\forall$, we choose a to be x_{i+1} where i is the maximum index of the free variables occurring in the conclusion of the rule.

More interesting is the question of whether the rules are *complete*, that is, do they suffice to prove every true proposition? We wish to show that, if a sequent s is true in all models, then we can prove it using our system of rules. Put another way, if we fail to prove s then it had better be false in at least one model. Given that we fail to prove s, how can we exhibit a model where s is false? When we attempt to prove s using our system of rules, if we eventually close every branch then we have a proof of s. So if we fail to prove s, there must be at least one branch which goes on forever. We call this branch a failing path, and denote it f. We can then define a model by taking $\mathbb{N}$ as the domain, interpreting each variable x_i as the number i, and interpreting an atomic predicate $P(x_{i_1}, \ldots, x_{i_n})$ as true iff $P(x_{i_1}, \ldots, x_{i_n})$ does not appear anywhere on the failing path f. By induction on the size of the formula, we can see that every formula appearing on f must get false. Since f starts at s, and all formulae in s get false, then s must also get false in this model. Let us consider the $\wedge$ case in the inductive argument. We must show that if $P \wedge Q$ appears on f, then $P \wedge Q$ gets false in the model. Assume $P \wedge Q$ appears on f. We must show that $P \wedge Q$ gets false in the model. Since $P \wedge Q$ appears in a sequent on f, eventually $P \wedge Q$ gets to the head of the sequent, and rule $\wedge$ is applied. At this point, either P appears on f or Q appears on f. By induction hypothesis, (at least one of) P or Q gets false in the model, so $P \wedge Q$ gets false in the model.

Having shown soundness and completeness for our system, we can derive an algorithm for first order validity: the algorithm simply takes the initial sequent s, and applies the rules of the logical system, keeping track of those sequents that appear at each stage. If at any stage all branches have been closed and there are no more sequents to consider, the algorithm will terminate with the answer *True*: if the algorithm terminates with *True*, then certainly we have a finite derivation and s is valid. Conversely, if s is valid, then by completeness there is a finite derivation, and we can argue that at stage n our algorithm has considered all potential derivations of depth less than or equal to n. In this way, we can see that the algorithm will terminate with *True* iff s is valid. We also note that by the undecidability result for first order logic, if the sequent would lead to an infinite derivation, the algorithm cannot always terminate with *False*.

3 Formalisation

3.1 Notation

We work in Isabelle/HOL, a variant of classical, simply typed, higher order logic. Isabelle/HOL has a meta-logic. The symbol $\Longrightarrow$ stands for the implication of the meta-logic. Nothing substantial is lost by considering this synonymous with the object level implication $\longrightarrow$. Nested meta-logical implication $A \Longrightarrow B \Longrightarrow C$ can be written $[\![A; B]\!] \Longrightarrow C$. The symbol $\equiv$ stands for the equality of the meta-logic. Again, nothing substantial is lost by considering this synonymous with the object level equality $=$. Type aliases are syntactic shorthand for the underlying type, and have no logical meaning. On the other hand, new types

may be introduced axiomatically, or defined in relation to an already existing type. The type $\mathbb{N}$ of natural numbers is written *nat*. Type constructors are functions mapping type lists to types. Application of a type constructor is typically written postfix. For example, the type of sets over an underlying type $'a$ is $'a\ set$. Application of a function f to an argument a is written simply $f\ a$. The function which takes an argument x and produces a result $f\ x$ is denoted by $\lambda x.\ f\ x$. The function which is exactly the same as f, except that x is mapped to y, is written $f(x := y)$. The type of a function with domain $'a$ and codomain $'b$ is $'a \Rightarrow 'b$. ML style datatypes are present, as is definition by primitive and wellfounded recursion. Lists are a particularly important datatype. The type of lists over a base type $'a$ is formed by applying the *list* type constructor, viz. $'a\ list$. The empty list is written $[]$, whilst the list xs with an additional x on the front is written $x \#\ xs$. The list containing $1, 2, 3$ is written $[1, 2, 3]$. The functions to take the head, *hd*, and tail, *tl*, of a list are as usual, such that $hd\ (x \#\ xs) = x$ and $tl\ (x \#\ xs) = xs$. The concatenation of two lists is written $xs\ @\ ys$. The function *set* takes an $'a\ list$ to an $'a\ set$ in the obvious way. The type of pairs over base types $'a$, $'b$ is written $'a \times 'b$, and has the standard projections *fst*, *snd*. Datatypes are accompanied by destructors in the form of case statements. For example, the case distinction on natural numbers may be written $case\ x\ of\ 0 \Rightarrow baseCase \mid Suc\ n \Rightarrow stepCase\ n$. This has the eta-contracted form $natcase\ baseCase\ stepCase$.

3.2 Formulas

Predicates P_i are identified by their index $i \in \mathbb{N}$. Similarly variables x_i.

```
types pred = nat
```

```
types var = nat
```

Terms are variables. Formulas are literals (positive and negative atomic predicates applied to tuples of variables, here represented as lists), conjunctions, disjunctions, foralls and exists. Variables and binding are handled by De Bruijn's nameless representation: a bound variable is a natural number indicating the number of enclosing quantifiers one must traverse to find the binding quantifier. This represents a deep embedding of the logic [WN04]. Free variables, substitution, and instantiation are defined as usual.

```
datatype form =
  PAtom pred (var list)
  | NAtom pred (var list)
  | FConj form form
  | FDisj form form
  | FAll form
  | FEx form
```

```
consts fv :: form ⇒ var list
primrec
  fv (PAtom p vs) = vs
  fv (NAtom p vs) = vs
  fv (FConj f g) = (fv f) @ (fv g)
  fv (FDisj f g) = (fv f) @ (fv g)
  fv (FAll f) = preSuc (fv f)
  fv (FEx f) = preSuc (fv f)
```

```
consts preSuc :: nat list ⇒ nat list
primrec
  preSuc [] = []
  preSuc (a#list) = (case a of 0 ⇒ preSuc list | Suc n ⇒ n#(preSuc list))
```

```
consts subst :: (nat ⇒ nat) ⇒ form ⇒ form
primrec
```

```
  subst r (PAtom p vs) = (PAtom p (map r vs))
  subst r (NAtom p vs) = (NAtom p (map r vs))
  subst r (FConj f g) = FConj (subst r f) (subst r g)
  subst r (FDisj f g) = FDisj (subst r f) (subst r g)
  subst r (FAll f) = FAll (subst (λ y. case y of 0 ⇒ 0 | Suc n ⇒ Suc (r n)) f)
  subst r (FEx f) = FEx (subst (λ y. case y of 0 ⇒ 0 | Suc n ⇒ Suc (r n)) f)


constdefs finst :: form ⇒ var ⇒ form
  finst body w ≡ subst (λ v. case v of 0 ⇒ w | Suc n ⇒ n) body
```

Sequents are formula lists. A numbered formula is a pair of a natural number and a formula. Numbered sequents are numbered formula lists. We define mappings between sequents and numbered sequents.

```
types seq = form list

types nform = nat × form

types nseq = nform list

constdefs s-of-ns :: nseq ⇒ seq
  s-of-ns ns ≡ map snd ns

constdefs ns-of-s :: seq ⇒ nseq
  ns-of-s s ≡ map (λ x. (0,x)) s
```

The free variables of a sequent, the maximum of a list of free variables, and a new variable are defined.

```
constdefs sfv :: seq ⇒ var list
  sfv s ≡ flatten (map fv s)

consts maxvar :: var list ⇒ var
primrec
  maxvar [] = 0
  maxvar (a#list) = max a (maxvar list)

constdefs newvar :: var list ⇒ var
  newvar vs ≡ Suc (maxvar vs)
```

3.3 Derivations

In addition to the rules in Fig. 1, we add the following rule to deal with the degenerate case of the empty sequent. We note that we could simply terminate the proof at this point, but that this rule ensures that the proofs in the rest of the mechanisation are uniform.

$$\frac{\vdash}{\vdash} \mathit{Nil}$$

Then to represent the rules, we use a function mapping the conclusion of a rule to a list of premises.

```
consts subs :: nseq ⇒ nseq list
primrec
  subs [] = [[]]
  subs (x#xs) =
  (let (m,f) = x in case f of
  PAtom p vs ⇒ if NAtom p vs ∈ set (map snd xs) then [] else [xs@[(0,PAtom p vs)]]
  | NAtom p vs ⇒ if (PAtom p vs) ∈ set (map snd xs) then [] else [xs@[(0,NAtom p vs)]]
  | FConj f g ⇒ [xs@[(0,f)],xs@[(0,g)]]
  | FDisj f g ⇒ [xs@[(0,f),(0,g)]]
  | FAll f ⇒ [xs@[(0,finst f (newvar (sfv (s-of-ns (x#xs)))))]]
  | FEx f ⇒ [xs@[(0,finst f m),(Suc m,FEx f)]])
```

Derivations are defined inductively wrt. this function. The additional natural number indicates the depth of a node in the derivation tree, and aids inductive

arguments. We also make an abstracting definition of a predicate to recognise a terminal sequent, that is, one that can be closed by an application of the rules $Ax, \overline{Ax}$.

```
consts deriv :: nseq ⇒ (nat × nseq) set
inductive deriv s
  intros
  init: (0,s) ∈ deriv s
  step: (n,x) ∈ deriv s ⟹ y ∈ set (subs x) ⟹ (Suc n,y) ∈ deriv s

consts is-axiom :: seq ⇒ bool
primrec
  is-axiom [] = False
  is-axiom (a#list) = ((∃ p vs. a = PAtom p vs ∧ NAtom p vs ∈ set list)
    ∨ (∃ p vs. a = NAtom p vs ∧ PAtom p vs ∈ set list))
```

Our first task is to show that these derivations are sound wrt. first order models.

3.4 Models

A first order model (M, I) is a set of elements M and an interpretation I of the syntactic predicates P_j as predicates $\mathcal{P}_j$ over tuples (represented as lists) of elements of the model. Which type should the elements of a model be drawn from? We assert the existence of a universal type[3].

```
typedecl U                    types model = U set × (pred ⇒ U list ⇒ bool)
```

An alternative would be to quantify over all models at all types. However, type quantification is currently not supported in HOL. We would like to echo Harrison [Har98] who notes the utility of type quantification in HOL [T.F92] in a similar context.

The third semantic notion is that of an environment, which is an assignment of elements in the model to free variables.

```
types env = var ⇒ U

constdefs is-env :: model ⇒ env ⇒ bool
  is-env MI e ≡ ∀ x. e x ∈ (fst MI)
```

Given a model and an environment, we can evaluate the truth of a formula, using standard Tarski semantics.

```
consts feval :: model ⇒ env ⇒ form ⇒ bool
primrec
  feval MI e (PAtom P vs) = (let IP = (snd MI) P in IP (map e vs))
  feval MI e (NAtom P vs) = (let IP = (snd MI) P in ¬ (IP (map e vs)))
  feval MI e (FConj f g) = ((feval MI e f) ∧ (feval MI e g))
  feval MI e (FDisj f g) = ((feval MI e f) ∨ (feval MI e g))
  feval MI e (FAll f) = (∀ m ∈ (fst MI). feval MI (λ y. case y of 0 ⇒ m | Suc n ⇒ e n) f)
  feval MI e (FEx f) = (∃ m ∈ (fst MI). feval MI (λ y. case y of 0 ⇒ m | Suc n ⇒ e n) f)
```

This extends to sequents, and finally we can say what it means for a sequent to be valid.

[3] This is one of only two places where we make axiomatic assertions. Both could be avoided by using existing type *ind* instead of declaring U.

```
consts seval :: model ⇒ env ⇒ seq ⇒ bool
primrec
  seval MI e [] = False
  seval MI e (x#xs) = (feval MI e x ∨ seval MI e xs)

constdefs svalid :: form list ⇒ bool
  svalid s ≡ ∀ MI e. is-env MI e ⟶ seval MI e s
```

3.5 Soundness

We prove that the rules are sound, that is, that any sequent at the root of a derivation is true in all models. Conceptually this is done by induction on the finite derivation, from leaf to root.

```
lemma soundness: finite (deriv (ns-of-s s)) ⟹ svalid s
```

For each rule, we need a lemma stating that if the premises are true in all models, then so too is the conclusion. We treat the most interesting case of the $\forall$ rule, which we prove for abitrary fresh u. This case depends on the following lemma, which states roughly that evaluating $[u/x]f$ in environment e is equivalent to evaluating f in an environment $e(x := e\ u)$, i.e. in the same environment except that the free variable x gets mapped to whatever u was mapped to by the original environment e.

```
lemma feval-finst: feval MI e (finst A u) = feval MI (nat-case (e u) e) A
```

The statement of the main lemma is as follows.

```
lemma sound-FAll: u ∉ set (sfv (FAll f#s)) ⟹ svalid (s@[finst f u]) ⟹ svalid (FAll f#s)
```

Suppose we wish to show that $\forall x.f$ is true in all models wrt. all environments e, and we assume that for u fresh wrt. f, $[u/x]f$ is true in all models wrt. all environments e'. To show $\forall x.f$ is true wrt. environment e, we must show that for all m, f is true wrt. environment $e(x := m)$. From the assumption, choosing e' to be $e(u := m)$, we get that $[u/x]f$ is true wrt. environment $e(u := m)$. By lemma *feval-finst*, this is equivalent to f being true wrt. environment $e(u := m)(x := (e(u := m)\ u))$, and this environment is simply $e(u := m)(x := m)$. Since u is fresh wrt. f, we actually have that f is true wrt. environment $e(x := m)$, which is what we had to show. This is the essential idea behind the proof of the main lemma. An Isar proof of this lemma is included in the development, but omitted here for space reasons.

3.6 Failing Path

We wish to show completeness of our rule system wrt. validity, i.e. if some sequent s is true in all models, then it is provable. Alternatively, if s is not provable, we must exhibit a model where s is false. If s is not provable, then when we attempt to prove it using the rules of our system, we will not end up with a finite derivation. If the derivation tree is infinite then, since it is finitely branching, we can use König's lemma to find an infinite path in the tree. We call this infinite path a failing path. We define a failing path as a function from a

derivation tree to a path through the derivation tree. Paths are simply functions with domain *nat*.

```
consts failing-path :: (nat × nseq) set ⇒ nat ⇒ (nat × nseq)
primrec
  failing-path ns 0 = (SOME x. x ∈ ns ∧ fst x = 0 ∧ infinite (deriv (snd x))
    ∧ ¬ is-axiom (s-of-ns (snd x)))
  failing-path ns (Suc n) = (let fn = failing-path ns n in SOME fsucn. fsucn ∈ ns
    ∧ fst fsucn = Suc n ∧ (snd fsucn) ∈ set (subs (snd fn)) ∧ infinite (deriv (snd fsucn))
    ∧ ¬ is-axiom (s-of-ns (snd fsucn)))
```

If f is the failing path for *deriv s* then the essential property of f is given in the following.

```
lemma (in loc1) is-path-f: infinite (deriv ns) ⟹ ∀ n. f n ∈ deriv ns ∧ fst (f n) = n
  ∧ (snd (f (Suc n))) ∈ set (subs (snd (f n))) ∧ infinite (deriv (snd (f n)))
```

Note that HOL is sufficiently powerful that we can simply define a failing path through a tree using the choice operator, avoiding an explicit invocation of König's lemma.

3.7 Contains, Considers

We now wish to talk about when a path f contains a numbered formula *nf* at position n. We also introduce the notion of when a formula is considered at a point n in a path, which is when the formula is at the head of the sequent at position n.

```
constdefs contains :: (nat ⇒ (nat × nseq)) ⇒ nat ⇒ nform ⇒ bool
  contains f n nf ≡ nf ∈ set (snd (f n))

constdefs considers :: (nat ⇒ (nat × nseq)) ⇒ nat ⇒ nform ⇒ bool
  considers f n nf ≡ case snd (f n) of [] ⇒ False | (x#xs) ⇒ x = nf
```

3.8 Models 2

A falsifying model will in general consist of at least countably many elements. So far, we have said nothing about the size of our universe type. We require that it is infinite so that it can contain an infinite falsifying model. We assert the existence of an injective function from *nat* to U[4].

```
consts ntou :: nat ⇒ U

axioms ntou: inj ntou
```

```
constdefs uton :: U ⇒ nat
  uton ≡ inv ntou
```

3.9 Falsifying Model from Failing Path

We are now in a position to define a falsifying model, given an infinite derivation.

```
constdefs model :: nseq ⇒ model
  model ns ≡ (range ntou, λ p ms. (let f = failing-path (deriv ns) in
  (∀ n m. ¬ contains f n (m,PAtom p (map uton ms)))))
```

[4] If we had taken the existing HOL type *ind* instead of declaring U, then we could avoid asserting this axiom, and our development would be conservative.

The point of the model is that any formula contained in a sequent on the failing path f gets false in the model. This is proved by induction on the size of the formula.

lemma ⟦ *f = failing-path (deriv (ns-of-s s)); infinite (deriv (ns-of-s s)); contains f n (m,A)* ⟧ ⟹ ¬ *feval (model (ns-of-s s)) ntou A*

Let us treat the case that $(\exists x.P\ x)^n$ appears on f. Then we know that $(\exists x.P\ x)^0$ appears on f. Then $(\exists x.P\ x)^0$ eventually gets considered, and $P\ x_0$ and $(\exists x.P\ x)^1$ appears on f. Then $(\exists x.P\ x)^1$ eventually gets considered, and $P\ x_1$ and $(\exists x.P\ x)^2$ appears on f. Continuing in this way, we see that for all n, $P\ x_n$ appears on f. Applying the induction hypothesis to each of these, we see that for all n, the interpretation of $P\ x_n$, i.e. $\mathcal{P}\ n$, is false in the model. So "there exists an n such that $\mathcal{P}\ n$" is false in the model, and so $\exists x.P\ x$ gets false in the model.

3.10 Completeness

Since the sequent s which gave rise to the infinite derivation appears on the failing path at position 0, and all formulas in s get false, it too must get false in the model. We have thus found our falsifying model, and s could never have been proved using any sound system of rules. The completeness lemma is as follows.

lemma *completeness*: *infinite (deriv (ns-of-s s))* ⟹ ¬ *svalid s*

3.11 Soundness and Completeness

We can combine our soundness and completeness results to get a lemma which connects validity and provability.

lemma *soundComplete*: *svalid s = finite (deriv (ns-of-s s))*

3.12 Algorithm and Computation

The rules of our system are deterministic. We therefore want to turn the rules into a deterministic algorithm. This algorithm checks to see if the derivation is finite by repeatedly applying the rules to all sequents at depth n, to obtain the list of sequents at depth $n+1$. If there are no sequents at a given depth, then all branches have been closed and we have found our finite derivation.

We define a global version of the algorithm as a function that takes an initial sequent s, and a number n, and gives back the list of sequents at depth n in the derivation rooted at s. We need a step function that takes a list of sequents at depth n in the derivation, and gives back a list of sequents at depth $n+1$.

constdefs *step* :: *nseq list* ⇒ *nseq list*
step ≡ λ *s*. *flatten* (*map subs s*)

We are going to iterate this step function repeatedly, so we need an iteration function.

consts *iter* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ — fold for nats
primrec
iter g a 0 = *a*
iter g a (*Suc n*) = *g* (*iter g a n*)

So iterating the step function n times on an initial sequent s gives the sequents at depth n in the derivation.

lemma $\forall\ x.\ ((n,x) \in deriv\ s) = (x \in set\ (iter\ step\ [s]\ n))$

Now we know that the derivation from s is finite iff there is some n such that there are no sequents at depth n in the derivation, in which case the nth iteration of the step function on s will be empty.

lemma *finite-deriv*: *finite* (*deriv s*) = ($\exists$ *m. iter step* [*s*] *m* = [])

Now the following is the definition in OCaML of a function that searches the natural numbers to see if there is an n such that the nth iteration of the *subs* function on an initial sequent s is empty. By lemma *finite-deriv*, if the derivation is finite, then there is some n such that the nth iteration is empty, and the algorithm will terminate. However, if the derivation is not finite, then our algorithm will fail to terminate, searching ever increasing n.

```
let rec prove' s n = (if iter step s n = [] then true else prove' s (n+1));;
let prove s = prove' [ns_of_s s] 0;;
```

There is an obvious source of inefficiency in that, having computed the nth iteration, we throw the result away and compute the $n + 1$ th iteration from scratch. The following is an equivalent, but much more efficient implementation.

```
let rec prove' s = (if s = [] then true else prove' (step s));;
let prove s = prove' [ns_of_s s];;
```

When we come to replicate this in HOL we run into a problem. HOL is a logic of total functions, whilst the general recursive OCaML definitions just given are clearly partial. However, we can improve our confidence in the OCaML definitions as follows. Firstly, we define our function *prove′* in a non-constructive way.

constdefs *prove′* :: *nseq list* ⇒ *bool*
prove′ s ≡ ∃ *m. iter step s m* = []

constdefs *prove* :: *seq* ⇒ *bool*
prove s ≡ *prove′* [*ns-of-s s*]

From lemma *finite-deriv*, *prove s* corresponds to the finiteness of the derivation from s.

lemma *finite-deriv-prove*: *finite* (*deriv* (*ns-of-s s*)) = *prove s*

Note that together with lemma *soundComplete* we have that validity of s is equivalent to *prove s*. We can even show that *prove'* satisfies the relation implied by the general recursive OCaML definition.

lemma *prove′*: *prove′ s* = (*if s* = [] *then True else prove′* (*step s*))

At this point, we believe that if the OCaML function terminates with *true*, then the derivation is finite. Conversely, we believe that if the derivation is finite, there is some n such that the nth iteration is empty, and our algorithm will discover this at stage n and terminate with *true*. On the other hand, we also believe that if the derivation is not finite, then the algorithm will fail to terminate. We have shown formally in HOL that any function that claimed to do this must satisfy the relation implied in lemma *prove'*. However, there are many functions that satisfy this recursive equation, such as the constant function of one argument that returns *true*, so that our beliefs have not been fully formalised. To go further would require some explicit notion of termination, and some way of handling general recursive definitions formally in HOL. We discuss this further in Sect. 5.

To execute the prover in Isabelle, we can rewrite the *prove* function to *prove'*, then use the lemma *prove'* to rewrite *prove'*. Further rewriting can occur. Examining the definitions, one sees that all functions involved are computable by rewriting. Of course, this rewriting will not terminate on some inputs, but if it does terminate with the output *True*, then the sequent is valid, and this will have been formally proved inside Isabelle. In terms of performance, one can comfortably run the verified prover inside Isabelle on small examples using the rewriting engine[5]. Alternatively, we have transported the definitions to OCaML, to give a directly executable program. For instance, the last few clauses of the OCaML program are as follows.

```
let subs t = match t with
    [] -> [[]]
  | (x::xs) -> let (m,f) = x in match f with
  PAtom (p,vs) -> if mem (NAtom (p,vs)) (map snd xs) then [] else [xs@[(0,PAtom (p,vs))]]
| NAtom (p,vs) -> if mem (PAtom (p,vs)) (map snd xs) then [] else [xs@[(0,NAtom (p,vs))]]
| FConj (f,g)  -> [xs@[(0,f)];xs@[(0,g)]]
| FDisj (f,g) -> [xs@[(0,f);(0,g)]]
| FAll f -> [xs@[(0,finst f (newvar (sfv (s_of_ns (x::xs)))))]]
| FEx f -> [xs@[(0,finst f m);(suc m,FEx f)]];;

let step = fun s -> flatten (map subs s);;

let rec prove' s = (if s = [] then true else prove' (step s));;

let prove s = prove' [ns_of_s s];;
```

4 Related Work

Completeness for first order logic has been mechanised several times, but without focusing on executability. In HOL Light, Harrison develops basic model theory for first order predicate logic in [Har98], mechanising a textbook by Kreisel and Krivine, but without addressing executability considerations. In Isabelle/HOL, Berghofer has also tackled classical first order predicate logic [Ber02], mechanising a textbook by Melvin and Fitting, again without focusing on executability.

In ALF, Persson has mechanised a constructive proof of intuitionistic predicate logic [Per96], but we were unable to trace the thesis. From comments in

[5] This applies to the newest version: the original was somewhat inefficient.

other work, it appears that Persson formalised the semantics of FOL formulae wrt. formal topology, so that the development is presumably substantially different from that here.

Completeness for propositional logic has also been tackled several times, although these results tend to be substantially easier to mechanise. There is a line of work by Underwood et al. in the NuPRL system. Primarily this involves mechanised proofs of completeness for intuitionistic propositional logic. The final paper in this line appears to be [Cal99], which usefully references much of the previous work. Included in this work is a paper [Und95] that discusses computational aspects of classical reasoning, applied to completeness results for intuitionistic logic, although these results were not mechanised. In Coq there has been much work on mechanising propositional logic. For instance, Weich tackles intuitionistic propositional logic in [Wei01].

The basic idea for this type of completeness proof is found in the work of Henkin [Hen49], where the author introduces the then radical idea of utilising the terms of the logical system as the elements of the model. Such a model is usually called a Henkin-model. The proof proceeds by successively extending the language and the term model until a maximally consistent and term complete model is formed. It is a relatively short step from here to defining a model directly from a failing derivation, as we have done here. However, we are unable to trace the paper in which this step appears for the first time. A succinct presentation of this approach is [WW92]. This work was originally mechanised in Isabelle/HOL by Margetson [Mar99]. This was then remechanised by Ridge, who modified the logical system, simplified and polished the proofs, and extended the work so that the prover is executable by rewriting inside Isabelle.

Less closely related is work on verifying proof checkers. For instance, Pollack has a verified type checker in [Pol95], and further work with McKinna is reported in [MP99].

More generally, this work is an exercise in mechanising results in metamathematics. A good example of a much more comprehensive mechanisation in this area is the work of Shankar on mechanising Gödel's incompleteness theorem [Sha94].

5 Conclusion and Future Work

We have presented a deterministic system of first order logic, proved soundness and completeness, and captured the system as an algorithm that can be directly executed inside Isabelle using the rewrite system. There are many extensions that might be considered.

Most immediately, we claim that if the derivation is finite, then the algorithm will terminate with *true*. To make this claim fully formal would require a treatment of general recursive definitions and explicit non-termination in HOL. One approach would be to work with a formalised semantics for OCaML, or some other suitable language. A more abstract approach would be to apply the techniques of LCF, a logic which explicitly deals with termination and recursive definitions.

Currently, there is a "leap of faith" [Har95] required to bridge the gap between the formalised definitions and those in OCaML. As we have shown, HOL is already a powerful language for expressing functional algorithms. A more radical approach than mechanising a theory of recursive functions inside HOL, is simply to carve out an exectuable subset of HOL itself. This is harder than it appears because one must treat general recursive definitions.

Terms in our system are simply variables. Although theoretically this is no restriction, it would be interesting to extend the mechanisation to deal with full first order terms. We might consider a representation of terms such as the following.

datatype *folterm = Var nat | App nat (folterm list)*

It is then not too difficult to enumerate these in an effective way. The next extension is to equality. Again, the lack of equality is theoretically no restriction, but it is usual to treat equality as a special relation, rather than axiomatising its properties.

We can also seek to extend the formalisation to cover other results in proof theory and automatic theorem proving, or as the basis of a more substantial verified theorem prover. Although we have not stated the result explicitly, our system does not use the *Cut* rule, so that the proofs generated are *Cut* free. *Cut* elimination is one of the main results of proof theory. Our proofs represent a semantic proof of *Cut* elimination. It would be interesting to tackle a syntactic proof, such as that by Pfenning [Pfe00]. Indeed, Pfenning cites *Cut* elimination as a challenge problem for formalisation. An approach along our current lines would have advantages over Pfenning's, in that the embedding is deeper, and properties such as coverage could be proven. Resolution is perhaps most easily treated as a variant of proof search in a system such as G3c [Avr93]. If syntactic *Cut* elimination were in place, a proof of completeness for a resolution based system would be relatively straight forward. Alternatively, one could try for a direct semantic proof. Mechanisation of these results could provide benefits to the community, allowing proposed improvements in algorithms to be formally assessed in terms of completeness preservation.

We alluded to the use of the reflection mechanism to incorporate verified code into the kernel of a theorem prover. It would be interesting to port the proofs to a system that supported such a feature, and investigate issues such as performance. Previous versions of this paper used the word "efficient" in the title, which referred to the fact that the algorithm was tail recursive, without backtracking. The algorithm does not use unification to select quantifier instantiations, and so is roughly comparable to Gilmore's procedure in terms of performance. In this sense, it is not competitive with current unification based approaches. It would be interesting to examine the performance of the system when extended to use unification.

The mechanisation described here can be found at the Archive for Formal Proofs [afp], which also includes the related OCaML code. Alternatively, the newest version is maintained at Ridge's homepage [Rid].

Finally, we would like to thank the anonymous reviewers for their extremely close reading which uncovered several inadequacies in a previous version.

References

[afp] The archive of formal proofs. http://afp.sourceforge.net/.

[Avr93] Arnon Avron. Gentzen-type systems, resolution and tableaux. *Journal of Automated Reasoning*, 10(2):265–281, 1993.

[Ber02] Stefan Berghofer. Formalising first order logic in isabelle, 2002. http://www4.in.tum.de/~streckem/Admin/club2_berghofer_model_theory.pdf.

[Cal99] James Caldwell. Intuitionistic tableau extracted. In *Proceedings of International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX'99)*, volume 1617 of *LNAI*, pages 82–96. Springer-Verlag, 1999. http://www.nuprl.org/documents/Caldwell/tableaux99.html.

[Har95] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz.

[Har98] John Harrison. Formalizing basic first order model theory. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lecture Notes in Computer Science*, pages 153–170, Canberra, Australia, 1998. Springer-Verlag.

[Hen49] Leon Henkin. The completeness of the first-order functional calculus. *The Journal of Symbolic Logic*, 14:159–166, 1949.

[Mar99] James Margetson. Completeness of the first order predicate calculus. 1999. Unpublished description of formalisation in Isabelle/HOL of [WW92].

[MP99] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4), November 1999.

[Per96] Henrik Persson. Constructive completeness of intuitionistic predicate logic. 1996. http://www.cs.chalmers.se/Cs/Research/Logic/publications.mhtml.

[Pfe00] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

[Pol95] Robert Pollack. A verified typechecker. In M.Dezani-Ciancaglini and G.Plotkin, editors, *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications, TLCA'95, Edinburgh*, volume 902 of *LNCS*. Springer-Verlag, April 1995.

[Qui62] Willard Van Orman Quine. *Mathematical Logic*. Harper and Row, 1962.

[Rid] Tom Ridge. Informatics homepage. http://homepages.inf.ed.ac.uk/s0128214/.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1994. http://www.csl.sri.com/users/shankar/goedel-book.html.

[T.F92] T.F. Melham. The HOL logic extended with quantification over type variables. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 3–18, Leuven, Belgium, 1992. North-Holland.

[Und95] Judith Underwood. Tableau for intuitionistic predicate logic as metatheory. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, volume 918 of *Lecture Notes in Artificial Intelligence*. Springer, 1995. http://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-321/.

[Wei01] Klaus Weich. *Improving Proof Search in Intuitionistic Propositional Logic*. Logos-Verlag, 2001. http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=767&lng=eng&id=.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223, pages 305–320, 2004.

[WW92] S. S. Wainer and L. A. Wallen. Basic proof theory. In S. S. Wainer, P. Aczel, and H. Simmons, editors, *Proof Theory: A Selection of Papers from the Leeds Proof Theory Programme 1990*, pages 3–26. Cambridge University Press, Cambridge, 1992.

[ZTA+] Richard Zach, Neil Tennant, Arnon Avron, Michael Kremer, Charles Parsons, and Timothy Y. Chow. The rule of generalization in fol, and pseudo-theorems. Thread on the FOM mailing list. http://www.cs.nyu.edu/pipermail/fom/2004-September/008513.html.

A Generic Network on Chip Model

Julien Schmaltz and Dominique Borrione

TIMA Laboratory, VDS Group, Joseph Fourier University,
46 avenue Felix Viallet, 38031 Grenoble Cedex, France
{Julien.Schmaltz, Dominique.Borrione}@imag.fr

Abstract. We present a generic network on chip model (named *GeNoC*) intended to serve as a reference for the design and the validation of high level specifications of communication virtual modules. The definition of the model relies on three independent groups of constrained functions: routing and topology, scheduling, interfaces. The model identifies the sufficient constraints that these functions must satisfy in order to prove the correctness of *GeNoC*. Hence, one can concentrate his efforts on the design and the verification of one group. As long as the constraints are satisfied the overall system correctness is still valid. We show some concrete instances of *GeNoC*. One of them is a state-of-the-art network taken from industry.

1 Introduction

To face the growth of integration capabilities (a few hundred million transistors with a 0.09 μm process), the *system-on-chip* (SoC) design paradigm has become familiar to embedded system manufacturers. This design methodology relies on the reuse of pre-existing processor and memory cores. In this framework, interconnections and the interoperability of the different components become the main design and verification challenges.

At the Register Transfer Level (RTL) and below, the design and the verification of a production quality SoC is well supported by numerous tools [19]. These tools are based on a combination of efficient methods to manipulate Boolean terms and reason in propositional logic: Boolean decision diagrams, SAT, automatic test pattern generation combined with structural analysis, partitioning and name matching [12].

To handle the complexity of a SoC, designs must start at a higher level of abstraction [25] and consider generic (*i.e. unbounded*) models. At this level, where emphasis is placed on the functional behavior, algorithmic techniques (e.g. Model Checking [15]) are not directly applicable because of their lack of abstract datatypes and the size explosion of their state-graph representation. Consequently, the initial design step is currently supported by simulation tools taking as input relatively *ad hoc* formalisms.

In this context, our objective is to provide formalisms and methodologies for the specification and the validation of parameterized communication architectures at their initial design steps. We consider *unbounded* systems where the

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 310–325, 2005.

number of interconnected nodes is an arbitrary, but *finite*, natural. Theorem proving systems, like ACL2 [14], PVS [17] or HOL [9], have reasoning capabilities which do not require fixed size models. They have been successfully applied to the verification of floating-point operators [22], microprograms [11,3,8] and pipelined machines [23,1]. If some of these systems have been applied to the verification of protocols [10,16,18,4], their application to on-chip communication hardware is new.

In this paper, we present a generic network on chip model intended to serve as a reference for the design and the validation of high level specifications of communication virtual modules. A function, named *GeNoC*, represents a generic communication architecture. It is defined in terms of three independent groups of key functions: routing and topology, scheduling, interfaces. We have identified constraints on the key functions that are sufficient to prove the overall correctness of *GeNoC*. Consequently, our proposed methodology to ensure that modules behave correctly in the overall system is modular. The definition and the identification of the properties (proof obligations) that ensure the correctness of the overall communication are the main contribution of this paper.

The next section presents an overview of *GeNoC*. We explain how we functionally represent communication architectures and we define our correctness criteria. In section 3, we detail the routing and the interconnect functions. We show that the routing of the Octagon network from *STMicroelectronics* and an XY routing algorithm in a 2D mesh are concrete instances of *GeNoC*. In section 4, we detail the generic scheduling function and we show two concrete instances of it: a circuit switched algorithm and a packet switched algorithm. Section 5 details the interfaces and exhibits a concrete example. In section 6, we present a precise definition of $GeNoC$ and how we prove it correct from the constraints on the key functions. Section 7 discusses related works. Finally, section 8 concludes the paper.

2 Overview of *GeNoC*

Our purpose is the modeling of communications between *points* (*e.g.* processors, co-processor, memories) of a network (Fig.1). Computations and communications are orthogonal, and are separated into applications and interfaces [21]. Each point of the network reflects this decomposition. This paper focuses on communications, applications are not considered further. To distinguish between interface-application and interface-interface communications, an interface and an application communicate using *messages*; two interfaces communicate using *frames*. In the examples of this paper, the need for the distinction between *messages* and *frames* is not obvious. But in the general case, the encapsulation of *messages* into *frames* with protocol information (*e.g.* Etherner, Bi-ϕ-M) produces objects of a quite distinct structure.

The points are not directly connected. A communication architecture, bus or network, determines how frames are transferred between them. The roles of this component for any communicating pair are:

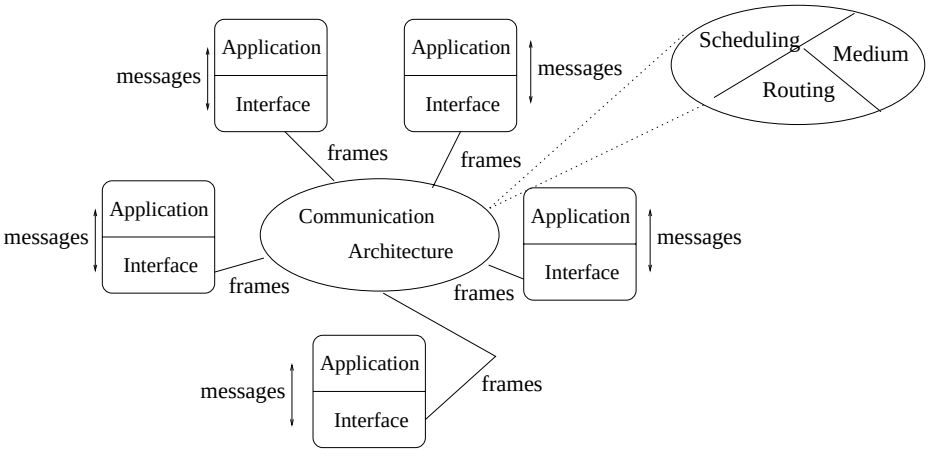


Fig. 1. Communications Principles

1. to compute the route between the points of a pair (routing)
2. to schedule or delay this communication according to the current traffic and the priority ordering between points (scheduling)
3. to convey the frame from one point to another (medium)

The scheduling policy constitutes one independent module. Routing and medium are intrinsically related by the topology of the network. They constitute another independent module.

The Generic Network on Chip (*GeNoC*) model is given Fig. 2. It represents a generic communication architecture. Our modeling style is functional: *GeNoC* is represented by a function, as well as each module. To show that our model can be extended we include interfaces in its definition. *GeNoC* takes as arguments the list of requested communications and the characteristics of the network. It produces two lists as results: the messages received by the destination of successful communications and the aborted communications. In the remainder of this section, we detail the basic components of the model.

The model makes no assumption on the domain *GenericNodeSet* used for referring to the points (or *nodes*) of the communication architecture. The set of nodes present in a particular architecture, denoted *NodeSet*, is generated by a network specific function *NodeSetGenerator(Params)*. The list *Params* represents the generating base for *NodeSet*. Function *ParamsHyps* is a network specific function which defines the list *Params*. Let *ValidNodesp(x)* be a predicate that defines the type of the nodes; the constraint on the generator is that it produces a list of valid nodes if *Params* satisfies its hypotheses. Consequently, for each instanciation of *GeNoC*, the following proof obligation will have to be relieved:

Proof Obligation 1. *NodeSet Definition*
$ParamsHyps(Params) \rightarrow ValidNodesp(NodeSetGenerator(Params))$

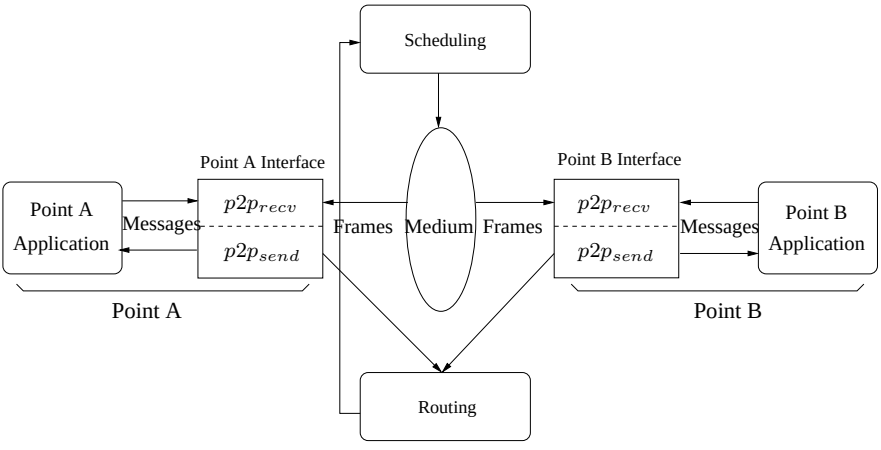


Fig. 2. The Generic Network on Chip

The main input of *GeNoC* is a list $\mathcal{T}$ of *transactions* of the form t = *(id A* msg_t *B)*. Transaction t represents the intention of the application A to send a message msg_t to the application B. A is the *origin* and B the *destination*. Both A and B are members of *NodeSet*. Each transaction is uniquely identified by a natural *id*. In a valid list, transactions appear in ascending order of the *ids*. Valid transactions are recognized by the predicate *Transactionsp(*$\mathcal{T}$*, NodeSet)*.

An interface communicates (*via* the transmission medium) with two components, an application and another interface. It is thus represented by two functions: $p2p_{send}$ computes a $frame$ from a *message*; $p2p_{recv}$ computes a *message* from a $frame$. The topology of the network, *i.e.* the physical interconnection of the different nodes, is represented by the function $Medium$. In this topology, the computation of routes is modeled by the function *Routing*. The scheduling policy of the architecture is represented by the function *Scheduling*. It splits a list of frames into a list of *scheduled* frames and a list of *delayed* frames.

Briefly, function *GeNoC* works as follows. For every message in the initial list of transactions, it computes the corresponding frame using $p2p_{send}$. Each frame together with its *id*, *origin* and *destination* constitutes a *missive*. A missive is valid if the ids are naturals (with no duplicate) and appear in ascending order; the origin and the destination are members of *NodeSet*. Valid missives are recognized by the predicate *Missivesp(x)*. Then, *GeNoC* computes the routes of the missives and schedules them using functions *Routing* and *Scheduling*. Once a route is computed, a *travel* denotes the list composed of a frame, its *id* and its route. Travels are valid if the ids are naturals (with no duplicate) and ordered. Valid travels are recognized by the predicate *TrLstp(x)*. The scheduled travels are executed by function $Medium$, and the corresponding results are computed by calling $p2p_{recv}$. The *delayed* travels are converted back to missives and constitute the argument of a recursive call to *GeNoC*. Function *GeNoC* halts if every attempt has been consumed. Its main output is the list of the results of the completed transactions.

Table 1. Terminology

Type of message	Purpose
message	information between applications and interfaces
frame	information between interfaces
NodeSet	set of existing nodes
missive	a frame with information about its travel ($Id\ origin\ frame\ destination$)
Missives	ordered list of missives
NodeLists	a set of ordered lists of existing nodes
route	a route in a network ($route \in NodeLists$)
origin	origin node of any message and frame
destination	destination node of any message and frame
Id	unique identifier of any message
att	a list of remaining attempts of every node
AttLst	domain of att
travel	a frame associated with its route and Id ($Id\ frame\ route$)
Travels	ordered list of travels
result	a tuple of the form ($Id\ result$)
$\mathcal{R}$	ordered list of results
transaction	tuple of the form ($Id\ origin\ message\ destination$)
$\mathcal{T}$	ordered list of transactions
$Params$	ordered list of the parameters of the system

GeNoC is a recursive function, and must be proved to terminate for two reasons. (1) It is a prerequisite for mechanized reasoning (in our case with the ACL2 system). To make sure that this function terminates, we associate to every point a *finite* number of attempts. At every recursive call of *GeNoC*, every point with a pending transaction will consume one attempt. The *association list att* stores the attempts and $att[i]$ denotes the number of remaining attempts of the node i. Function *SumOfAttempts(att)* computes the sum of the remaining attempts of the nodes and is used as the decreasing measure of parameter *att*. (2) The main reason why termination must be ensured is that $\mathcal{T}$ represents the transactions and is *finite*. If *GeNoC* never terminates on finite inputs, this means that at least one transaction is processed forever, *i.e.* there exists a deadlock.

Let *AttLst* be the domain of *att*, *GeNoC* has the following functionality:

$$GeNoC:\ \mathcal{P}(\mathcal{T}) \times GenericNodeSet \times AttLst \mapsto \mathcal{P}(\mathcal{R})\ \times \mathcal{P}(\mathcal{T})$$

Transactions may not run to completion (*e.g.* due to network contention). The second output list of *GeNoC* is named *Aborted* and contains the cancelled or lost transactions. The first output list $\mathcal{R}$ contains the results of the completed transactions. Every result r is of the form *(id* msg_r*)* and represents the reception of a message msg_r by its final destination B.

Function *GeNoC* is considered correct if every non aborted transaction $t =$ *(id A msg B)* is completed in such a way that B effectively receives msg. We thus

need to prove that for every final result r, there is a unique initial transaction t such that t has the same id and msg as r.

Proof Obligation 2. *GeNoC Result*

$$\forall(id_r\ msg_r) \in \mathcal{R},\ \exists!(id_t\ a_t\ msg_t\ b_t) \in \mathcal{T} \mid id_r = id_t \wedge msg_r = msg_t$$

This formula must be complemented by a condition $TravelCondition$ (defined later) which proves that every transaction is received by the correct destination. The correctness criterion for the function $GeNoC$ is to prove both the $TravelCondition$ and *GeNoC Result* under the following hypotheses:

$$TheHyps = Transactionsp(\mathcal{T}, NodeSet) \wedge ParamsHyps(Params) \\ \wedge NodeSet = NodeSetGenerator(Params)$$

The complete terminology used in this paper is summarized in Table 1. In the next sections, we detail the definition of every key function and present the sufficient constraints that the key functions must satisfy in order to prove the final correctness of *GeNoC*.

3 Topology and Routing

The expected topology is defined by predicate $CheckRoutep(route, NodeSet)$ which holds if any two successive nodes in *route* are adjacent nodes in the topology. Predicate $Availablemovesp(TrLst, NodeSet)$ checks that $CheckRoutep$ holds for the route of every travel of $TrLst$ and provides the relation between functions $Medium$ and *Routing*. Both $Medium$ and *Routing* must be shown to conform to the desired topology.

3.1 Topology

Function $Medium$ represents the physical node interconnection. It has the following functionality:

$$Medium : \mathcal{P}(Travels) \times GenericNodeSet \mapsto \mathcal{P}(Travels)$$

It must be an accurate model of the topology. If every frame route is consistent with the topology, function $Medium$ moves frames from one node to another without modifying them. Otherwise, the result of function $Medium$ must differ from the initial travel list indicating an error. Formally, function $Medium$ is a projection of its first dimension if and only if *Availablemovesp* holds. This is expressed by the following proof obligation:

Proof Obligation 3. *Medium Constraint*

$$TrLstp(TrLst) \\ \rightarrow \\ Medium(TrLst, NodeSet) = TrLst \leftrightarrow Availablemovesp(TrLst, NodeSet)$$

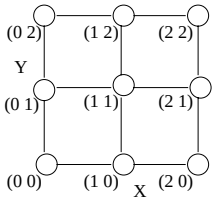


Fig. 3. 2D Mesh

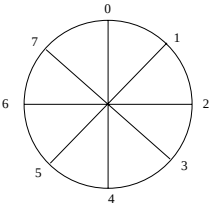


Fig. 4. Octagon

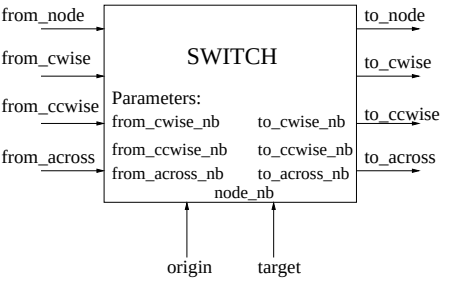


Fig. 5. Generic Switch

Example 1. **Octagon Topology.** Let us consider the Octagon [13] topology given in Fig. 4. The eight nodes are connected by bidirectional links, but we can extend the topology to an arbitrary number of nodes Num_Nodes that is a multiple of four. We define $Params$ to be Num_Node. Function *OctagonParamsHyps(Num_Node)* is equal to $Num\_Node \bmod 4 = 0 \wedge Num\_Node \in \mathbb{N}$. In the Octagon, the set of existing nodes is equal to the naturals up to $Num\_Node - 1$. Function $OctagonNodeGen$ generates the nodes of the Octagon. We prove that *Proof Obligation 1* holds for $OctagonParamsHyps$ and $OctagonNodeGen$. In the Octagon topology, three moves are available: clockwise, counterclockwise or across. These moves define the predicate $OctagonAvailableMovesp$.

The nodes are connected using 4x4 switches (Fig. 5). In our functional paradigm, we represent a generic switch component by a function $Switch$. It takes as arguments: the four inputs (*from_x*), two commands (*origin* and *target*) and switch specific parameters. It produces a new value for every output. The switch reads a frame on the input selected by the value of *origin*, and writes the frame on the output selected by the value of *target*. The other outputs are set to a default "no frame" value. In our model, a frame travels on its route r as a result of iterative calls to function *Switch*, until every node of r has been visited. Let i be the current node at a given travel step in route r. *Switch* is called with i as *node_nb*. *origin* and *target* take the previous and next node numbers w.r.t. i in r. If i is the first node of r, *origin* is equal to i. If i is the last node of r, *target* is equal to i. The values assigned to the outputs of *Switch*, as a result of executing one travel step, are used in the next call to *Switch* where i is replaced by its successor in r. These calls to *Switch* represent the structure of the interconnected nodes effectively involved in the travel along route r. The function $OctagonMedium$ is then defined by executing the travel of every element of its input argument $TrLst$. The proof of *Medium Constraint* [24] showed that our example is a valid instance of the generic medium model.

Example 2. **2D Mesh Topology.** We model the mesh topology in a similar way to the Octagon example. Here the parameters $Params$ are the size of the X and Y dimensions. Predicate *Paramsp(Params)* recognizes such parameters. A node is a coordinate and the set of existing nodes is generated by the function *MeshNodeGenerator(Params)*. Fig. 3 shows the coordinates for $X = 3$ and $Y =$

2. In this example, a list of valid coordinates is recognized by the predicate *ValidCoordinatesp(x)*. The first theorem that we prove is that *Proof Obligation 1* holds for these functions.

Let $(x_i\ y_i)$ and $(x_{i+1}\ y_{i+1})$ be two successive nodes of a route. This step is considered available if $x_{i+1} = x_i \pm 1 \wedge y_{i+1} = y_i$ or $x_{i+1} = x_i \wedge y_{i+1} = y_i \pm 1$. The predicate *MeshAvMovesp(TrLst, NodeSet)* recognizes such routes.

We model a generic 5x5 switch (4 inputs and 4 outputs for the X and Y dimensions, 1 input and 1 output for the node itself) by a function $MeshSwitch$. A frame travels on its route by iterative calls to this function. The 2D mesh is represented by the function *Mesh(TrsLst, NodeSet)*. We prove that function $Mesh$ is a valid instance of the generic $Medium$ function.

3.2 Routing

Function *Routing* represents the routing algorithm of the network. It takes as arguments: the list of missives and the set of existing nodes. It returns a travel list that associates to every missive its route in the topology. The functionality of *Routing* is:

$$Routing : \mathcal{P}(Missives) \times GenericNodeSet \mapsto \mathcal{P}(Travels)$$

A route from an *origin* to a *destination* is correct if (a) it contains no duplicate; (b) its first element is the origin; (c) its last element is the destination; (d) it is a subset of $NodeSet$ and (e) it is consistent with the network topology. The last condition is dependent on the topology. It is represented by the network specific predicate *Availablemovesp(TrLst, NodeSet)*. On the contrary, the first conditions are common to all networks. One of the main concerns in a routing algorithm is the absence of deadlocks. In the case of deterministic routing algorithms, it has been proven [7] that the absence of cycles in routes is a necessary and sufficient condition to prevent deadlocks. The first condition ensures this property. The predicate *CorrectRoutesp(TrLst, Missives, NodeSet)* checks that every route in $TrLst$ of the corresponding missive in $Missives$ satisfies the properties (a) to (d) above. Function *Routing* is correct if it satisfies both *CorrectRoutesp* and *Availablemovesp*. The main proof obligation about the routing algorithm defines the $TravelCondition$:

Proof Obligation 4. *Routing Correctness - TravelCondition*

$$\begin{array}{l} Missivesp(Missives, NodeSet) \\ \rightarrow \\ CorrectRoutesp(Routing(Missives, NodeSet), Missives, NodeSet) \\ \wedge Availablemovesp(Routing(Missives, NodeSet), NodeSet) \end{array}$$

Another important constraint states that the routing function creates a travel list without modifying the content (id, frame, origin and destination) of the initial transactions. Let *ToMissives(TrLst)* be a function that builds a list of missives from a list of travels. It grabs the id, the frame of each travel and for every created missive the origin and the destination are the first and the last element

of the route of the travel. If we convert the resulting travel list into a list of missives, we obtain the initial input $Missives$.

Proof Obligation 5. *Routing Missives*
$Missivesp(Missives, NodeSet)$
$\rightarrow$
$ToMissives(Routing(Missives, NodeSet)) = Missives$

Example 3. **Routing in the Octagon.** Let us consider the Octagon network [13] (Fig. 4). The routing of a packet is accomplished as follows. Each node compares the tag ($Packet\_addr$) to its own address ($Node\_addr$) to determine the next action. The node computes the relative address of a packet as:

$$Rel\_addr \ = \ (Packet\_addr \ - \ Node\_addr) \ mod \ 8 \tag{1}$$

At each node, the route of packets is a function of Rel_addr as follows:

- $Rel\_addr = 0$, destination node has been reached
- $Rel\_addr = 1$ or 2, route clockwise
- $Rel\_addr = 6$ or 7, route counterclockwise
- route across otherwise

This routing algorithm is represented for an arbitrary number Num_Node of nodes by a function $Octagon\_Route$ (Num_Node is a multiple of 4). This function computes the path - a list of node numbers - between the origin and the destination nodes. In a previous study [24] we proved that $Octagon\_Route$ satisfies the *TravelCondition*. We prove that *Proof obligation* 5 holds for $Octagon\_Route$. Consequently, this shows that $Octagon\_Route$ is a valid instance of function $Routing$.

Example 4. **XY-Routing in a 2D mesh.** Dimension-order routing [7] is a deterministic routing scheme well-suited for uniform traffic distribution. The dimensions of the network are arranged in predetermined monotonic order and frames traverse dimensions in sequence. Frames first traverse the network in the lowest or highest dimension until no further move is needed in this dimension. Then, they go along the next dimension and so forth until they reach their destination. Dimension-order routing in two-dimensional meshes is called XY routing (Fig. 3). The dimensions of the mesh are named X and Y. Frames travel along the X dimension completely and then along the Y dimension.

We define a function *XYRouting(missives, NodeSet)* which represents the dimension order algorithm in the 2D mesh. Because frames never travel in reverse direction of the dimension ordering, there is no duplicate in routes (*i.e.* no cycle) and thus no deadlock. We prove that $XYRouting$ satisfies $CorrectRoutesp$ and $MeshAvMovesp$. Finally, we can prove that *Proof Obligation* 4 holds for $XYRouting$. This function associates a route to a $frame$ without modifying it and therefore *Proof Obligation* 5 holds. This shows that the dimension order routing in a 2D mesh is a valid instance of the generic routing function of $GeNoC$.

4 Scheduling

The scheduling policy of the network is represented by function *Scheduling*. It takes as arguments the travel list produced by the routing function, the set of existing nodes and the list of attempts *att*. It returns a list of *scheduled* travels, a list of *delayed* travels, and updates *att*. The functionality of *Scheduling* is:

$$Scheduling : \mathcal{P}(Travels) \times GenericNodeSet \times AttLst \mapsto \mathcal{P}(Travels)^2 \times AttLst$$

For readability, we shall identify the codomain of *Scheduling* as:

$$Scheduled \times Delayed \times AttLst$$

Let us consider a function $f : X_1 \times X_2 \mapsto X_3 \times X_4$. We will note by $f_{\downarrow_{X_3}}(x_1, x_2)$ and $f_{\downarrow_{X_4}}(x_1, x_2)$ the projections of f on X_3 and X_4.

To ensure the termination of *GeNoC*, one attempt must be consumed at each call. The first constraint on *Scheduling* is:

Proof Obligation 6. *Scheduling Consumes At Least one Attempt*
Let natt be $Scheduling_{\downarrow_{AttLst}}(TrLst, NodeSet, att)$ *in*
$SumOfAttempts(att) \neq 0$
$\rightarrow SumOfAttempts(natt) < SumOfAttempts(att)$

Let $TrLst/ids$ denote a sublist of the list of travels $TrLst$ which is the result of filtering $TrLst$ according to some ids.

Example 5. If $TrLst$ is ((123 m_1 (1 3 9)) (212 m_2 (12 4 25)) (313 m_3 (1 12 3))), then $TrLst/(123\ 313)$ is ((*123 m1 (1 3 9)*) (*313 m3 (1 12 3)*))

The delayed travels are converted back to missives in the recursive call of *GeNoC*. These missives must be a sublist of the initial missives. Filtering $TrLst$ according to any subset *ids* of the identifiers del_{ids} of the delayed travels, must produce the same list as filtering the delayed travels according to *ids*.

Proof Obligation 7. *Delayed Travels Correctness*
Let del be $Scheduling_{\downarrow_{Delayed}}(TrLst, NodeSet, att)$ *in*
$\forall ids \subseteq \mathbb{N}.NoDuplicatesp(ids) \wedge ids \subseteq del_{ids} \wedge TrLstp(TrLst)$
$\rightarrow\ del/ids = TrLst/ids$

Concerning the scheduled travels, they must contain the same frame as in the initial travel list. In contrast to *Delayed Travels Correctness*, we only prove that filtering the initial $TrLst$ according to the ids $sched_{ids}$ of the scheduled travels produces the scheduled travels. In the final proof, this constraint is sufficient to prove the correctness of *GeNoC*.

Proof Obligation 8. *Scheduled Travels Not Modified*
Let sched be $Scheduling_{\downarrow_{Scheduled}}(TrLst, NodeSet, att)$ *in*
$TrLstp(TrLst)$
$\rightarrow$
$sched = TrLst/sched_{ids}$

The scheduling policy may modify the routes of frames (*e.g.* in case of network contention). But, the routes chosen by the scheduling policy must still be correct.

Proof Obligation 9. *Scheduling Preserves Route Correctness*
$Let\ sched\ be\ Scheduling_{\downarrow Scheduled}(TrLst, NodeSet, att)\ in$
$CorrectRoutesp(TrLst, ToMissives(TrLst), NodeSet)$
$\wedge Availablemovesp(TrLst)$
$\rightarrow$
$Correctroutesp(sched, ToMissives(TrLst)/sched_{ids}, NodeSet)$
$\wedge Availablemovesp(sched)$

The goal of the scheduling policy is to split a list of travels in two distinct sublists. A last proof obligation ensures that the intersection of the scheduled travels and the delayed travels is empty.

Example 6. **Circuit and Packet Switched Scheduling.** Two modes are commonly used in SoC communication architectures: circuit switched and packet switched. In the circuit switched mode, a complete path is allocated for each transaction. A circuit algorithm has already been studied [24] and is easily proved to be fully compliant with *GeNoC*. In this mode, a travel list contains non-overlapping communications if a node is used in at most one route.

The packet switched mode is more complex and allows more concurrency. In this mode, a message *msg* is divided into packets. In our formalism, this is expressed by generating a transaction for each packet. For instance, the transactions ($(123\ 12\ P_0\ 24)$ $(124\ 12\ P_1\ 24)$) represent the emission of two packets at node 12. The destination of these packets is node 24. In a packet switched network, after each hop from a node A to a node B, A is available for other packets. We have modeled a packet algorithm inspired from the Octagon's [13]. Let *prev* be a list of lists of nodes. $prev[i]$ denotes the list of nodes used by all scheduled travels at the i*th* hop. An additional travel is scheduled if at every hop number i, the i*th* node of its route does not belong to $prev[i]$ and if every node of its route has at least one attempt left. This condition is defined as the function *PackCond(route, prev, att)* to be:

$$\bigwedge_{i=0}^{len(route)-1} route[i] \cap prev[i] = \emptyset \wedge AttLst[route[i]] \neq 0$$

Function *PacketScheduling* represents this algorithm and has the same functionality as function *Scheduling*:

$$PacketScheduling : \mathcal{P}(Travels) \times GenericNodeSet \times AttLst \mapsto Scheduled \times Delayed \times AttLst$$

Its definition is as follows:

Definition 1. *Packet Scheduling*
PacketScheduling(TrLst, NodeSet, att) $\triangleq$
if *SumOfAttempts(att) = 0* **then**

list(∅, TrLst, att)
else
Let (*sched del*) **be** *PacketScheduler(TrLst, att, ∅, ∅, ∅)* **in**
list(sched, del, ConsumeAttempts(att))
endif

where *PacketScheduler* is:

Definition 2. *Packet Scheduler*
PacketScheduler(TrLst, att, sched, prev, del) $\triangleq$
if $TrLst = \emptyset$ **then**
list(Rev(sched), Rev(del))
else
if $PackCond(Route(Head(TrLst)), prev, att)$ **then**
PacketScheduler(Tail(TrLst), AttLst, $Head(TrLst) \cap sched$*,*
UpdatePrev(prev, route), del)
else
PacketScheduler(Tail(TrLst), att, sched, prev,
$Head(TrLst) \cap del$*)*
endif
endif

Function *UpdatePrev* properly updates the list *prev* by inserting the nodes of *route* at the corresponding sublist.

Before trying to check that function *PacketScheduling* is a valid instance of *Scheduling*, we prove the algorithm correct *per se*. This correctness is achieved if for any hop a node is used at most once. Let $ExtractHops(TrLst)$ be the function which creates the lists of the nodes used in every hops; and let *steps* denote $ExtractHops(PacketScheduling_{\downarrow Scheduled}(TrLst,\ NodeSet,\ att))$, the correctness of *PacketScheduling* is expressed by the following formula:

$$\bigwedge_{i=0}^{len(steps)-1} NoDuplicatesp(steps[i]) \tag{2}$$

Now, the algorithm captures effectively our initial intent. Because we want to validate *PacketScheduling* without considering other functions, we define the predicate *PackAv* to return "true" for every possible values of its input arguments. We use *PackAv* as an instance of $AvailableMovesp$. We prove that every proof obligation holds on *PacketScheduling* under the proper substitutions. This shows that both circuit and packet modes are valid instances of the generic *Scheduling* function and that *Scheduling* constitutes an independent function.

5 Interfaces

Function $p2p_{send}$ takes as input a *message* and outputs a $frame$.

$$p2p_{send} : Messages \mapsto Frames$$

Function $p2p_{recv}$ takes as input a $frame$ and outputs a $message$.

$$p2p_{recv} : Frames \mapsto Messages$$

The constraint on the interface functions is that their composition yields the identity function. This is formally expressed by the following proof obligation:

Proof Obligation 10. *Transfer Constraint*
$\forall\ m \in Messages,\ p2p_{recv} \circ p2p_{send}(m) = m$

Example 7. In his study of asynchrony, Moore [16] represents a sender and a receiver by two functions: *send* and *recv*. He proves that their composition is an identity. Therefore, his functions are valid interfaces. The interfaces can be validated as an independent group of functions.

6 Definition and Correctness of *GeNoC*

Let $ComputeMissives$ denote the function that applies function $p2p_{send}$ recursively and produces a list of missives from the list of transactions $\mathcal{T}$. Let $ComputeResults$ denote a similar function that recursively applies $p2p_{recv}$ to produce the list of results $\mathcal{R}$. We have the following definition for *GeNoC*:

Definition 3. *GeNoC*
GeNoC($\mathcal{T}$*, NodeSet, att)* $\triangleq$
Let (*Responses Aborted*) **be**
GeNoC$_t$*(ComputeMissives(*$\mathcal{T}$*), NodeSet, att,* $\emptyset$*)* **in**
list(ComputeResults(Responses), Aborted)

where *GeNoC*$_t$ is the following function:

Definition 4. *GeNoC*$_t$
GeNoC$_t$*(Missives, NodeSet, att, Responses)* $\triangleq$
if $SumOfAttempts(att) = 0$ **then**
list(ToMissives(Responses), Missives)
else
Let(*sched del att*$_1$) **be**
Scheduling(Routing(Missives,NodeSet),NodeSet,att) **in**
Let *sched*$_1$ **be** *Medium(sched, NodeSet)* **in**
GeNoC$_t$*(ToMissives(del), NodeSet, att*$_1$*,*
sched$_1 \cup$ *Responses)*
endif

As stated in section 2, *GeNoC* must satisfy both the $TravelCondition$ and *GeNoC Result*. The $TravelCondition$ is satisfied by the routing algorithm and only *GeNoC Result* remains unproved. In the context of our work, *GeNoC Result* does not translate directly in the ACL2 logic, because ACL2 is quantifier free. As a walk around, we extract the messages from the projection of the input transactions $\mathcal{T}$ according to the ids $\mathcal{R}_{ids}$ of the output $\mathcal{R}$, and we extract the results of $\mathcal{R}$. These two lists must be equal. Formally, we prove the following theorem:

Theorem 1. *GeNoC Correctness*
Let $\mathcal{R}$ *be* $GeNoC_{\downarrow_{P(R)}}(\mathcal{T}, NodeSet, att)$ *in*

$$TheHyps \rightarrow Messages(\mathcal{T}/\mathcal{R}_{ids}) = Results(\mathcal{R})$$

Proof. The proof is done by induction on $\mathcal{T}$ and the induction hypothesis is obtained by the substitution suggested by the recursive call of $GeNoC_t$. The base case is trivial. For the induction step, thanks to proof obligations 3, 4 and 9, we can remove the call to function $Medium$. Then, because of proof obligations 5, 7 and 8, every result is processed by the composition of $p2p_{send}$ with $p2p_{recv}$. This composition is proved to be the identity function by proof obligation 10. □

7 Related Works

The very active field of protocol verification best relates to the modeling of interfaces in *GeNoC*. A huge body of work is based on the use of temporal logics. The temporal properties are checked using model checkers (*e.g.* [5,6]) or a combination of model checkers and theorem provers (*e.g.* [4]). For instance, Roychoudhury *et al.* [20] use the SMV model checker to debug a RTL implementation of the AMBA AHB protocol from ARM. The AMBA protocol family was then studied by Amjad [2]. He uses a model checker implemented in HOL to verify two different protocols. Then, he uses the HOL theorem proving system to verify their composition. These studies are complementary to our work: they focus on behavioral aspects of low level models rather than functionalities of abstract entities. In closer frameworks, the idea of Proof Obligation 10 is present in many previous works (*e.g.* [16]). For instance, in the recent work of Pike *et al.* [18] concerning fault-tolerant distributed systems, this property is formalized using relations instead of functions. Our work is also related to the study of deadlock free algorithms (*c.f* section 3.2). One originality of *GeNoC* is to consider both protocols and routing algorithms in a single framework.

8 Conclusions and Future Work

In this paper, we have presented a generic model for deterministic networks on chip named *GeNoC*. This model relies on a functional formalism for communications. *GeNoC* models a *complete* communication system and its correctness includes the proof that messages are either lost or eventually reach their expected destination without being modified. The correctness also ensures that the routing algorithm is deadlock free. The model identifies key functions and the sufficient constraints they must satisfy in order to prove the correctness of *GeNoC*. The key functions are separated in independent groups. Hence, one can concentrate his efforts on the design and the verification of one group. As long as the constraints are satisfied the overall system correctness is still valid.

The *GeNoC* ACL2 file contains around 1 500 lines, 40 functions and 85 theorems. We show two different applications of this generic model: an XY routing

scheme in a 2D mesh and the Octagon network from *STMicroelectronics*. We also show that the packet and circuit switched scheduling policies are instances of *GeNoC*. Most of the human effort was spent on identifying the sufficient constraints ensuring that they lead to the proof of the final theorem.

Our model has been developed using the ACL2 theorem proving system. Some features were really helpful. We take advantage of the executability of the ACL2 logic to debug failed proofs and definitions. We reach the genericity using the *encapsulation principle*. Using the *functional instanciation* feature, the proof obligations that must be satisfied to prove that a function is a valid instance of a generic function are automatically generated. Nevertheless, our functional style and our generic model are general and should be processable by other theorem proving systems.

Acknowledgements

The first author would like to thank Warren Hunt and J Strother Moore for fruitful discussions.

References

1. M. Aagaard: A Hazards-Based Correctness Statement for Pipelined Circuits. In *CHARME 2003*, LNCS, Vol. 2860, D. Geist and E. Tronci, eds. Springer-Verlag, (2003).
2. H. Amjad: Model Checking the AMBA Protocol in HOL. Technical Report, University of Cambridge, Computer Laboratory, (September 2004).
3. S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach and W. J. Paul: Instantiating Uninterpreted Functional Units and Memory System: Functional Verification of the VAMP. In *CHARME 2003* pp. 51-65, LNCS, Vol. 2860, D. Geist and E. Tronci, eds. Springer-Verlag, (2003).
4. R. Bharadwaj, A. Felty and F. Stomp: Formalizing Inductive Proofs of Network Algorithms. In *Proc. of the 1995 Asian Computing Conference*, LNCS, Vol. 1023, Springer-Verlag, (1995).
5. E.M. Clarke, O. Grumberg and S. Jha : Verifying Parameterized Networks. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5):726-750, (1997).
6. E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long and K.L. McMillan: Verification of the Futurebus+ Cache Coherence Protocol. In *Proc. of CHDL'93*, pp 15-30, (1993).
7. W.J. Dally and C.L. Seitz: Deadlock Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, vol. 36, no. 5, pp. 547-553, (May 1987).
8. A.C.J. Fox: Formal Specification and Verification of ARM6. In *TPHOLs'03*, pp 25-40, LNCS, Vol. 2758, D. Basin and B. Wolff, eds, Springer-Verlag (2003).
9. M.J.C. Gordon and T.F. Melham (Eds): Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press (1993).
10. K. Havelung and N. Shankar: Experiments in Theorem Proving and Model Checking for Protocol Verification in *Proc. of Formal Methods Europe (FME'96)*, LNCS 1051 (1996).

11. W.A. Hunt, Jr: Microprocessors Design Verification. *Journal of Automated Reasoning*, 5(4):429-460, (1989).
12. J. Jain, K. Mohanram, D. Moundanos, I. Wegener and Y. Lu: Analysis of Composition Complexity and How to Obtain Smaller Canonical Graphs. In *Proceedings of IEEE-ACM Design Automaction Conference (DAC'00)*, pp. 681-686, (2000)
13. F. Karim, A. Nguyen, S. Dey and R. Rao : On-Chip Communication Architecture for OC-768 Network Processor. *Design Automation Conference* (2001)
14. M. Kaufmann, P. Manolios and J Strother Moore: Computer-Aided Reasoning: An Approach. Kluwer Academic Publisher (2000)
15. K. L. McMillan: Symbolic Model Checking. *Kluwer Academic Publishers*, (1993).
16. J Strother Moore: A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol, *Formal Aspects of Computing* (1993)
17. S. Owre, J.B. Rushby and N. Shankar: PVS: A Prototype Verication System. In D. Kapoor (ed.),*11th International Conference on Automated Deduction (CADE).* Volume 607 of LNAI., Springer-Verlag, pp 748–752 (1992).
18. L. Pike, J. Maddalon, P. Milner and A. Geser: Abstractions for Fault-Tolerant Distributed System Verification. In *TPHOLs'04*, pp 257-270, LNCS, Vol. 3223, Springer-Verlag (2004).
19. W. Roesner: What is Beyond the RTL Horizon for Microprocessor and System Design. Invited Speaker. In *CHARME 2003*, LNCS, Vol. 2860, D. Geist and E. Tronci, eds. Springer-Verlag, (2003).
20. A. Roychoudhury, T. Mitra and S.R. Karri: Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol. In *Design Automation and Test Europe (DATE'03)*, pp 828-833 (2003).
21. J. A. Rowson and A. Sangiovanni-Vincentelli: Interface-Based Design. *Design Automation Conference* (1997)
22. D. Russinoff: A Mechanically Checked Proof of IEEE Compliance of a Register Transfer Level Specification of the AMD-K7 Floating-Point Multiplication, Division and Square Root Instructions. *London Mathematical Society Journa of Computation and Mathematics*, 1:148-200, (December 1998).
23. J.Sawada and W.A. Hunt, Jr: Processor Verification with Precise Exceptions and Speculative Execution. In *CAV 98*, volume 1254 of LNCS, Springer-Verlag (1998).
24. J. Schmaltz and D. Borrione: A Functional Approach to the Formal Specification of Networks on Chip. *In. Proc. of Formal Methods in Computer-Aided Design (FMCAD'04)*, A.J. Hu and A.K. Martin (Eds), LNCS 3312, Springer-Verlag, pp 52-66, (2004).
25. G. Spirakis: Beyond Verification: Formal Methods in Design. Invited Speaker. *Formal Methods in Computer-Aided Design (FMCAD04)* LNCS 3312 (2004)

Formal Verification of a SHA-1 Circuit Core Using ACL2

Diana Toma and Dominique Borrione

TIMA Laboratory, VDS Group, Grenoble, France
{diana.toma, dominique.borrione}@imag.fr

Abstract. Our study was part of a project aiming at the design and verification of a circuit for secure communications between a computer and a terminal smart card reader. A SHA-1 component is included in the circuit. SHA-1 is a cryptographic primive that produces, for any message, a 160 bit message digest. We formalize the standard specification in ACL2, then automatically produce the ACL2 model for the VHDL RTL design; finally, we prove the implementation compliant with the specification. We apply a stepwise approach that proves theorems about each computation step of the RTL design, using intermediate digest functions.

1 Introduction

The SHA-1 is a standardized hash function [1], which processes a message up to 2^{64} bits, and produces a 160 bit message digest, with the following property: any alteration to the initial input message will result, with a very high probability, in a different message digest. The applications of this algorithm include fast encryption, password storage and verification, computer virus detection, etc. The study of SHA-1 was motivated by a cooperative project[1] with industrial partners, aiming at the design and verification of a circuit for secure communications between a computer and a terminal smart card reader. Security considerations were at the heart of the project, it was thus of utmost importance to guarantee the correctness of the system components dedicated to security. For the SHA-1 component, formal methods were applied both for the validation of the functional specification, and for the verification of the implementation.

The SHA-1 is an iterative algorithm, involving a large number of repetitions over an arbitrary number of 512-bit blocks. Property verification by model checking was first attempted, and provided some correctness statements about the internal design synchronization. But more powerful methods had to be applied to establish that, whatever the length of the input message, the digest is computed according to the standardized algorithm. It was thus decided to apply mechanized theorem proving technology. We chose the ACL2 logic, for its high degree of automation, and reusable libraries of function definitions and theorem proofs [4]. Moreover, the input model, being written in a subset of Common Lisp,

[1] This work was supported by ISIA2 contract granted by the French Ministry of Industry (DIGITIP).

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 326–341, 2005.

is both executable and provable. Before investing human time in a proof, it is thus possible to check the model on test vectors, a common simulation activity in design verification which helps debug the formal model and gain designer's confidence in it. This feature was key to showing the compliance of our specification model with the SHA standard document, since numeric data only is provided as validation test.

2 SHA-1 Specification

2.1 The SHA-1 Algorithm

The principle of the SHA-1 is shown on Figure 1. The input message M, a bit sequence of arbitrary length $L < 2^{64}$, undergoes two preprocessing steps:

- Padding: M is concatenated by bit 1, followed by k bits 0, followed by the 64-bit binary representation of number L. k is the least non-negative solution to the equation: (L+1+k) mod 512 = 448. As a result, the padded message holds on a multiple of 512 bits.
- Parsing: The padded message is splited in blocks of 512 bits.

The computation of the message digest is an 80-iteration algorithm over each message block, in order; a block is viewed as a sequence of 32 bit words, which are selected and combined with the contents of five 32-bit internal registers (A, B, C, D, E), using XOR and shift operations. At the start of the computation, the internal registers are initialized with predefined constants $\boldsymbol{H_0} = (H_0, H_1, H_2, H_3, H_4)$. At the end of each block processing, they contain the digest obtained so far. This digest is used as an initial value for processing the next block, if there is one.

According to the SHA-1 standard [1], the digest phase operates on the 16 words W_i $(0 \leq i \leq 15)$ of a padded block in order to generate 80 words. The SHA-1 algorithm is formalized in ACL2 and the detailed model can be found in [2]. Because of hardware efficiency constraints, the VHDL design implements an alternative digest algorithm presented in the standard, which stores only

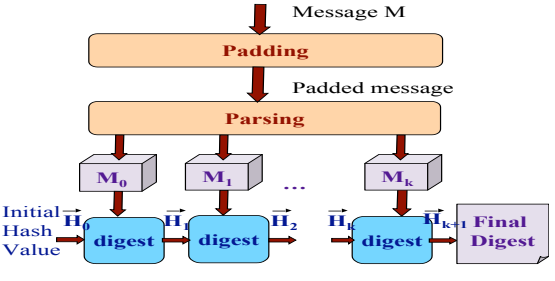


Fig. 1. Secure Hash Algorithm

sixteen W words and not eighty. We summarize below the principle of the alternative algorithm, knowing that we have already proven that the two methods are equivalent.

W_j $(0 \leq j \leq 79)$ and the five main variables A, B, C, D, E, are computed in the same loop, and each W_j starting from j=16 is written in place of W_j mod 16. The first sixteen words are made of the padded block itself. The 64 remaining words and A, B, C, D, E are generated as follows, where $ROTL^n$ indicates a n-bit circular left shift operation:

```
for j=0 to 79 do
        s=j∧MASK;
        if j≥16 then
          W_s=ROTL^1 (W_{(s+13)∧MASK} XOR W_{(s+8)∧MASK}
                      XOR W_{(s+2)∧MASK} XOR W_s);
        endif;
        TEMP=ROTL^5(A) + F_j (B, C, D) + E + W_s + K_j;
        E=D; D=C; C=ROTL^30(B); B=A; A=TEMP;
endfor.
```

where MASK = '00000000000000000000000000001111', F_j are functions and K_j constants defined in the SHA-1 standard.

When one block has been processed, the values of H_i are updated:

$H_0 = H_0$+A; $H_1 = H_1$+B; $H_2 = H_2$+C; $H_3 = H_3$+D; $H_4 = H_4$+E;

The computation continues until there are no more blocks to process. The last H_i updated values represent the message digest of the initial message.

2.2 The Functional Model

Here is the corresponding ACL2 model. Lisp syntax has been replaced by an algorithmic notation, for readability.

Digest-one-block-Spec, computes the variables A, B, C, D, E for $80 - j$ steps of the algorithm, starting from step j. Let ABC_Vars $\stackrel{def}{=}$ *List* (A, B, C, D, E)

```
Digest-one-block-Spec (j, ABC_Vars, Block) def=
  if Natural (j) then
    if 80 ≤ j then ABC_Vars
      else let New-Block be
                  if 16 ≤ j then Replace (s (j), Word_Spec (j, Block), Block)
                    else Block fi
             New-ABC_Vars be
                  ⟨Temp_Spec (j, ABC_Vars, New-Block), A, Rotl (30, B), C, D⟩
             in
             Digest-one-block-Spec (j+1, New-ABC_Vars, New-Block)
    fi
    else nil
fi
```

In function *Digest-one-block-Spec* above, $Word\_Spec$ computes the word $W_{s(j)}$ during step j of the algorithm, $Temp\_Spec$ computes the variable TEMP for the step j of algorithm, s (j) computes j *mod* 16 and *Replace* (i, W, Block) replaces the i-th word of block with W.

Word_Spec (j, Block) $\stackrel{def}{=}$
Rotl-spec (1, *B-Xor* (Block[*B-And* (*mask*, 13 + *s* (j))],
Block[*B-And* (*mask*, 8 + *s* (j))],
Block[*B-And* (*mask*, 2 + *s* (j))],
Block[*B-And* (*mask*, j)]))

Temp_Spec (j, ABC_Vars, Block) $\stackrel{def}{=}$
Rotl-spec (5,A) + *F-spec* (j, B, C, D) + E + Block[*s*(j)] + *K*(j)

s (j) $\stackrel{def}{=}$ *Bv-nat-be* (*B-And* (*Nat-bv-be* (j), *mask*))

In the above *Bv-nat-be*, *Nat-bv-be* are conversion functions according to the big endian representation, *B-And* and *B-Xor* are macros computing the logic *and*, respectively *xor* operation between two bits, and between two bit-vectors of possibly distinct lengths. + is addition overloaded with the addition between two bit-vectors, and between a natural and a bit-vector, *Rotl-spec (n, w)* rotates to left the bit-vector w with n bits.

The function *Digest-Spec* computes the digest for a padded message:

Digest-Spec (Message, Hash_Val) $\stackrel{def}{=}$
if *empty* (Message) **then** Hash_Val
else *Digest-Spec* (*Cdr* (Message),
Update (Hash_Val,
Digest-one-block-Spec (0, Hash_Val, *Car* (Message))))
fi

When the last block has been processed, the message digest contains the last values of $Hash\_Val$. Otherwise, one block is digest, the result is updated, and the computation continues for the rest of the message.

Sha_Spec defines the specification for SHA-1:

Sha_Spec (Message) $\stackrel{def}{=}$
Digest-Spec (*Parsing* (*Padding* (Message, 512)), Initial_Hash_Val)

An arbitrary message m is first padded, then it is parsed into blocks of 512 bits. The list of blocks is digested with the algorithm constants as initial hash values: Initial_Hash_Val $\stackrel{def}{=}$ ⟨*h0*, *h1*, *h2*, *h3*, *h4*⟩.

2.3 Validation of the Formal Functional Specification

The SHA-1 ACL2 model is executed on the test benches given in the standard document to check that the returned result is as expected. A complementary validation is obtained by proving the mathematical properties of the algorithm, using ACL2 [2]. In fact, a more general model has been written, to capture the common principles of the four versions of the SHA algorithm: SHA-1, SHA-256, SHA-384 and SHA-512, which differ essentially in the sizes of the message blocks, word, and digest. Seventy function definitions and over a hundred lemmas were written. Among the safety theorems that were proven for the SHA-1:

- The length of the padded message is a non-zero multiple of 512.
- The last 64 bits of the padded message represent the binary coding of the length.
- The first L bits of the padded message represent the initial message.
- The bits between the end-of-the-message bit and the last 64 bits are all 0.
- After parsing the padded message, the result is a vector of blocks, each of 512 bits.
- The final result of the SHA-1 is a five 32-bit words message digest.

Due to the nature of the digest computation, there is no straightforward algebraic expression for it; thus, the validation of the specification consisted in showing properties of the result of each processing step rather than proving its equivalence with a mathematical function.

3 SHA-1 Implementation

3.1 Main Characteristics of the VHDL Design

The SHA-1 design provided by our project partners implements only the digest computation and it takes as input the message already padded from an external RAM. The RAM is also used to store the modified W_s computed during the digest. The VHDL model is written at the RTL level [6] and is fully synchronous. The pin description of the core is given in Table 1. The design also has 23 internal memories. We will refer only to some of them, which seem important to illustrate our approach: a, b, c, d, e are 32-bit registers storing the digest computation, *state* is a 3-bit vector giving the state of the control automaton, $blocks\_left$ is a 6-bit word representing the number of blocks that remain to be processed, *count* is a 8-bit word counting the iterations of the digest.

The SHA core is composed of a control machine and a data path. The data path contains a compact description of the operators necessary for the digest computation. The transition graph of the control automaton for the state machine is shown on Figure 2.

Table 1. Pin description of the SHA core

start, reset	input	bit	start signal, asynchronous core reset
reset_done	input	bit	invalidates the output *done*
clk	input	bit	clock signal
rdata	input	32-bit	Input data
base_address	input	12-bit	first word W RAM address
nb_block	input	6-bit	number of blocks
address	output	12-bit	RAM address
ram_sel, ram_write	output	bit	RAM select and write signals
wdata	output	32-bit	computed W to store into the RAM
busy	output	bit	core busy by digest computation
done	output	bit	digest message available
aout, bout, cout, dout, eout	output	32-bit	message digest output

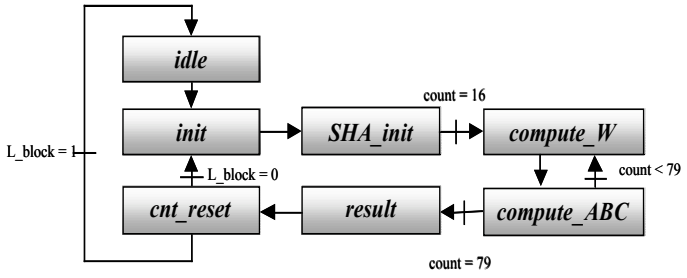


Fig. 2. VHDL control automaton

The global behaviour is the following:

- *idle* is the wait state;
- *init* loads the constants H_0 to H_4 into the registers that store A, B, C, D, E;
- *SHA_init* reads one block from the RAM and computes the first sixteen values of A, B, C, D, E, i.e. the first sixteen steps of the algorithm;
- *compute_W* computes one of the 64 remaining words, W ;
- *compute_ABC* computes the values of A, B, C, D, E corresponding to the previous W and updates W in the RAM.
  The states *compute_W* and *compute_ABC* are repeated 64 times;
- *result* adds the last values of A, B, C, D, E to the last values of H_0 to H_4 respectively;
- *cnt_reset* resets the different counters; when the last block has been processed (L_block = 1) then signal *done* is set to 1, indicating that the message digest is available.

The RAM has the following behaviour:

- if the *ram_sel* bit is 1, the RAM is enabled, i.e. reading or writing is allowed;
- writing is allowed only if the *ram_write* bit is 1, and in this case the data *wdata* is written at *address*
- if the *ram_write* bit is 0, reading is allowed, and the data from *address* is put in *rdata* which is one of the inputs for the design.

3.2 Formal Model of the SHA-1 Design into ACL2

The VHDL is automatically translated into a functional model using a method based on symbolic simulation developed by our team [5]. The model is simulated symbolically for one clock cycle, actually corresponding to several VHDL simulation cycles, to extract one transition function per output and state variable of the design. The body of a transition function is an if_expression, an arithmetic or a Boolean expression. The functions are translated into Lisp and used to define the Moore machine for the initial VHDL description.

A state of the Moore machine is the set of all internal registers and all the outputs of SHA, grouped in vector LOCALS. A step is modeled as a function *Sim_Step* which takes as parameters the inputs of SHA and LOCALS at clock cycle k, and which produces LOCALS at clock cycle k+1 (k is a natural).

$$Sim\_Step \text{ (Input, LOCALS)} \stackrel{def}{=} \langle NextSig\_s \text{ (Input, LOCALS)} \rangle_{s \in St}$$

$NextSig\_s$ denotes the transition function for signal $s \in$ LOCALS. There is one such function for each $s \in$ LOCALS, obtained by simbolic simulation. The body of *Sim_Step* is the composition of all the $NextSig\_s$.

The external memory, RAM, must also be added to the state as it stores the message during the computation.

We also extract from the VHDL the types for all input, internal and output objects. These informations are translated into ACL2 as predicates.

Finally, the circuit is defined by the function *Sha_Vhdl* which has two parameters: the sequence of inputs *List_Inputs* and the state *St*. *St* is the concatenation of LOCALS and RAM.

St $\stackrel{def}{=}$ (LOCALS | RAM)

Sha_Vhdl (List_Inputs, St) $\stackrel{def}{=}$
if *empty* (List_Inputs) **then** St
else let* New_LOCALS **be**
Sim_Step (*Read-RAM* (LOCALS, RAM) | *Car* (List_Inputs), LOCALS)
New_RAM **be** *Write-RAM* (New_LOCALS, RAM)
in
Sha_Vhdl (*Cdr* (List_Inputs), New_LOCALS | New_RAM)
fi

The length of *List_Inputs* gives the number of clock cycles, and *List_Inputs* represents the list of symbolic or numeric values for the SHA-1 input ports at each clock cycle. If the inputs list is empty, the computation is finished and *Sha_Vhdl* returns the state *St*. Otherwise, the next state is computed by calling the step function *Sim_Step*, then LOCALS and RAM are updated. Again, this model is executable, and we have initially checked it using the test benches provided in the SHA standard.

The following *Sha_Vhdl* property allows the combination of different behaviors of the circuit:

Lemma. $\forall$ List_inputs a valid list of inputs, $\forall$ i, j naturals,
$i + j \leq Length$ (List_Inputs)
Implies
Sha_Vhdl (*First* (i+j, List_Inputs), St) =
Sha_Vhdl (*First* (i, List_inputs),
Sha_Vhdl (*First* (j, *Last*(i, List_inputs)), St))

Table 2. The symbolic input for *Sha_Vhdl*

Cycle	**1**	**2**	**3**	...
Input	*input_cycle_1*	*input_cycle_2*	L_Input	
Reset	1	0	0	...
Start	X	1	X	...
Reset_done	X	X	X	...
Nb_block	X	nb_block	nb_block	...
Base_address	X	X	base_address	...

The expected environment for *Sha_Vhdl* is described in Table 2, where X stands for "don't care", *nb_block* is the 6-bit representation of the natural number *nb*: the number of blocks to be processed, *base_address* is a bit-vector of size 12. *nb_block* and *base_address* are symbolic. The fist two cycle initialize the design and start the computation. Starting from the third cycle, reset is fixed to 0, and *nb_block* and *base_address* signals are stable. Thus, the proof will be done for:

$$\text{List\_Inputs} \stackrel{def}{=} (\langle \text{input\_cycle\_1, input\_cycle\_2} \rangle \mid \text{L\_input}).$$

The RAM is modeled as a list of couples $\langle address$, 32-bit word$\rangle$ where *address* is symbolic. In order to preserve the generality of the proof, we model the RAM as being the concatenation of two RAMs: one RAM of no interest for the computation and the second RAM starting from *base_address*. The addresses domains of the two RAMs are disjoint.

4 Proof of Correctness of the Implementation

At this point, we have two models of SHA-1 in ACL2: the translation by hand of the standard *Sha_Spec* which has no timing information, and the automatic translation of the VHDL description *Sha_Vhdl* which is clock cycle accurate. Because *Sha_Vhdl* takes as input an already padded message, contrary to *Sha_Spec* which inputs the initial message, we will compare *Sha_Vhdl* with *Digest-Spec* instead of *Sha_Spec*.

Thus, we must show this correctness statement: for an arbitrary input message, the execution of *Sha_Vhdl* for the appropriate input and the appropriate number of clock cycles (until the computation is done) returns the same message digest as the one returned by *Digest-Spec*.

Sha_Vhdl needs 3 clock cycles to initialize the system and set A, B, C, D, E to their initial values; then it needs 342 clock cycles to compute the digest for one block. The 342 cycles are decomposed as: 16 for reading the first 16 words and computing 16 steps of digest, 320 to compute an intermediate digest, 3 to combine the results with the initial hash values of the block, 2 to store the message digest obtained so far. The last cycle returns to the digest computation for the next block, or to the *idle* state. So, in order to process *nb* blocks, the design needs $3 + (342 * nb)$ clock cycles.

The above could let the reader believe that we are performing simulation. This is not the case. The reasoning engine considers the initial value of all memo-

ries and registers as arbitrary, and *nb* (the number of blocks) to be an unbounded (but finite) natural integer.

4.1 Defining the Intermediate Functions

One difficulty in performing the proof arises because there is no way to write a relation between *Digest-Spec* and the result of the VHDL computation. Clearly *Digest-Spec* involves a time abstraction with respect to *Sha_Vhdl* but the two models are not directly equivalent. To solve the problem, we define intermediate digest functions having a similar construction as the specification functions but with design elements as input: *Digest-one-block-Impl* for the computation of the digest for one block, and *Digest-Impl* for the overall digest computation. Actually, each intermediate function corresponds to an abstraction of the behavior of the VHDL state machine.

The *Digest-one-block-Impl* computes i steps of the algorithm starting from step *count*.

```
Digest-one-block-Impl
      (i, count, a, b, c, d, e, RAM, base_address, nb_block, blocks_left) def=
  if Is_zero (i) then List (a, b, c, d, e)
    else let* Address be
                 New-address (nb_block, blocks_left, base_address, count, 0)
              Computed-Word be
                 Word-Impl (count, base_address, nb_block, blocks_left, RAM)
              Word-from-RAM be if 16 ≤ count then Computed-Word
                                    else Get(Address, RAM) fi
              New-a be Temp-Impl (count, a, b, c, d, e, Word-from-RAM)
              New-RAM be if 16 ≤ count then
                              Put (Address, Computed-Word, RAM)
                              else RAM fi
              in
              Digest-one-block-Impl (i-1, count+1, New-a, a, Next-b (b), c, d,
                              New-RAM, base_address, nb_block, blocks_left)
  fi
```

At each step the current RAM address of $W_{count \bmod 16}$ is computed. If $count < 16$ then $W_{count \bmod 16}$ is read from the RAM, otherwise it is computed by xor-ing words from the RAM, (Computed-Word), and the RAM is updated. The values of a, b, c, d, e are then updated and the computation continues until i is 0.

In the function *Digest-one-block-Impl* above, *Temp-Impl* computes the intermediate variable TEMP, *New-address* computes the address of $W_{i+count \bmod 16}$ relative to *base_address*, *Word-Impl* computes the new $W_{count \bmod 16}$, *Get* reads a word from the RAM, and *Put* writes a word into the RAM.

```
Temp-Impl(count, a, b, c, d, e, data) def=
          F-impl (count, b, c, d) + e + data + Rotl-Impl (5, a) + K (count)
```

```
New-address (nb_block, blocks_left, base_address, count, i) =def
            base_address + i + Segment (4, 8, count) +
            16 * (nb_block - blocks_left)

Word-Impl (count, base_address, nb_block, blocks_left, RAM) =def
  Rotl-Impl (1, B-Xor(
        Get (New-address (nb_block, blocks_left, base_address, count, 0), RAM),
        Get (New-address (nb_block, blocks_left, base_address, count, 2), RAM),
        Get (New-address (nb_block, blocks_left, base_address, count, 8), RAM),
        Get (New-address (nb_block, blocks_left, base_address, count, 13), RAM)))
```

F-impl and *Rotl-Impl* are the logic function F and rotation left operation ROTL defined in the VHDL implementation. The arithmetic operations + and - are overloaded for two bit-vectors and for a bit-vector and a natural.

Another difficulty in performing the proof is brought by the presence of the RAM in the implementation and by the added circuitry to access the informations and write back partial results during the computation. Moreover, a 512-bit block is overwritten by the W_i, $(16 \leq i \leq 79)$ words during the digest computation. In contrast, the specification is a functional algorithm that processes each message block without side-effect. Thus, we needed to define a function modeling how the RAM is modified during computation: *Modified-RAM*.

While $count < 16$ there is no writing operation on the RAM. If $16 \leq count$, at each step *count* of the algorithm, the word $W_{count \bmod 16}$ is updated in the RAM.

```
Modified-RAM (i, count, RAM, base_address, nb_blocks, blocks_left) =def
   if Is_zero(i) then RAM
     elsif 16 ≤ count then
          Modified-RAM (i-1, count+1,
               Put (New-address (nb_blocks, blocks_left, base_address, count, 0),
                    Word-Impl (count, base_address, nb_blocks, blocks_left, RAM)
                    RAM)
               base_address, nb_blocks, blocks_left)
          else Modified-RAM(i-1, count+1, RAM, base_address,
                                nb_blocks, blocks_left) fi fi
```

The general implementation digest function *Digest-Impl* returns the resulting hash values if there are no more blocks left to process. Otherwise, it computes the digest for one block, updates the hash values and then continues the computation for the rest of the message stored in the modified RAM.

```
Digest-Impl (Hash_Val, RAM, base_address, nb_blocks, blocks_left) =def
  if Is_zero (blocks_left) then Hash_Val
    else Digest-Impl (
           Update (Hash_Val,
                   Digest-one-block-Impl (80, '00000000', Hash_Val,
                     RAM, base_address, nb_blocks, blocks_left))
           Modified-RAM (80, '00000000', RAM, base_address, nb_blocks, blocks_left)
                        base_address, nb_blocks, Minus(blocks_left,1))
  fi
```

Digest-one-block-Impl and *Modified-RAM* have a similar decomposition property which allows resoning about the design behaviour:

$\forall$ *i*, *j* naturals
Digest-one-block-Impl
(i+j, count, a, b, c, d, e, RAM, params) =
Digest-one-block-Impl(i, count+j,
Digest-one-block-Impl (j, count, a, b, c, d, e, RAM, params)
Modified-RAM (j, count, RAM, params), params)

$\forall$ *i*, *j* naturals,
Modified-RAM (i+j, count, RAM, params) =
Modified-RAM (i, count+j,
Modified-RAM (j, count, RAM, params), params)

4.2 VHDL Design Behavior vs Intermediate Functions

To prove properties about the VHDL design behavior we developed a stepwise approach, which proves intermediate theorems for each main computation step of the overall Sha_Vhdl. A computation step corresponds to a state of the VHDL state machine (Figure 3).

- Theorems 1 to 3 establish the results of the initialization phases;
- Theorem 4 corresponds to the first 16 computation steps. The RAM is unchanged;
- Theorem 5 corresponds to the subsequent 64 steps: the block is overwritten in the RAM;
- Theorems 6 and 7 update the block digest and initialize the computation for the next block;
- Theorem 8 combines Theorems 3 to 7 to establish the result of 342 clock cycles over one block;
- Theorem 9 combines Theorems 1,2 and 8 to establish the result of the VHDL computation for nb blocks (over 3+342*nb cycles).

Some of the theorems above (Theorems 4, 5, 8, 9) prove that the result of the VHDL design, for a given number of clock cycles, is equal with the intermediate functions. These theorems are proved using the same method: the computation step is first generalized for an arbitrary number j of clock cycles and then proved by induction on j and on the circuit state, to be compliant with the intermediate function. Here is the general induction scheme constructed for this purpose:

1. Base case: $Is\_zero(j)$ or $empty(input)$ or $empty(st) \Rightarrow P(j, input, st)$
2. Induction step:
$P(j-1, Last(step, First(j * step, input)), Sha\_Vhdl(First(step, input), st)) \Rightarrow$
$P(j, input, st)$

where P is the property to prove, j is the number of clock cycles and *step* represents the number of clock cycles needed for the design to compute one

algorithm step. For instance the VHDL needs 5 clock cycles to compute one of the last 64 steps of the digest algorithm, or 342 clock cycles to compute the digest for a block message. After the generalized theorem is proved, j is instanciated to the actual number of cycles performed by the circuit (16 for SHA_init, 320 for looping between *compute_w* and *compute_ABC*, etc.). All proofs details in the ACL2 logic can be found in [3].

Theorem 1. *(From an arbitrary state to idle)*
Starting from an arbitrary state, after one cycle, with input_cycle_1 as input, the system is in the idle state and the RAM is not modified.

Theorem 2. *(From idle to init)*
Starting from state idle, after 2 cycles, the system is in state init, the state variables a, b ,c, d, e are initialized with the initial hash values which are constants of the standard, blocks_left is initialized with the number of block to be processed, i.e. nb_block, all the other state variables are initialized and the RAM is not modified.

Theorem 3. *(From init to SHA_init)*
Starting from state init, after 1 cycle the system is in the SHA_init state (i.e. the computation can begin), count is set to 0 and the RAM has not changed.

The theorems above are obtained by symbolic execution of Sha_Vhdl.

Theorem 4. *(From SHA_init to compute_W)*
Starting from the initial computation state, after 16 cycles, RAM is not modified, the system state is compute_W, count is '00010000', and a, b, c, d, e hold the result of the first 16 steps of the digest computed by Digest-one-block-Impl.

To prove Theorem 4 we first prove a more general form: the generalization is done for *count* and the number of clock cycles. The new theorem is proved by induction using the scheme defined above, with $step = 1$. Then, j is instantiated with 16 and *count* with '00000000'.

Theorem 5. *(From compute_W to result)*
Starting from the computation state for the first word, after 320 cycles, RAM is modified, the system state is result, count is '01010000', and a, b, c, d, e hold the result of the last 64 steps of the digest computed by Digest-one-block-Impl.

The 5*64=320 cycles are needed to compute the digest of a block: 5 cycles for the computation of TEMP (the intermediate variable) and the algorithm must be applied 64 times to compute the intermediate digest.

As in the previous case, a generalized version of the theorem is proved first. The parameters to be generalized are the same: *count* and the number of clock cycles. The new theorem is proved by induction using the general scheme, with $step = 5$. Then, j is instantiated with 64 and *count* with '00010000'.

Some lemmas are needed to compute the behaviour of Sha_Vhdl for 5 cycles, starting from *compute_W* state:

Lemma. (From *compute_W* to *compute_ABC*)

Starting from any word computation state (state = *compute_W*), after 4 cycles, the corresponding $W_{count\ mod\ 16}$ word is computed by *Word-Impl*, the new state is *compute_ABC*, *count* and RAM are unchanged.

Lemma. (From *compute_ABC* to *compute_W* or *result*)

Starting from any variables computation state (state = *compute_ABC*, *count* $\leq$ 79), after one cycle if *count* is 79 then the algorithm finished and the new state is *result*, otherwise, the algorithm was applied less than 79 times and the new state is *compute_W*. In both cases *count* is incremented, TEMP is computed and $W_{count\ mod\ 16}$ is written in RAM at its corresponding address.

Theorem 6. *(From result to cnt_reset)*
Starting from the result state, after 3 cycles, the system is in state cnt_reset, the number of blocks to be processed is decremented and a, b, c, d, e are added to h0, h1, h2, h3, h4, which are intended to hold the hash values during the computation.

Theorem 7. *(From cnt_reset to init or idle)*
Starting from the reseting state, after 2 cycles, if the number of blocks to be processed is higher than 0, then the new state is init and count is reset to '00000000', otherwise the new state is idle, done is 1 and the values of a, b, c, d, e are available as output.

Theorem 8. *(From init to init or idle $\Leftrightarrow$ One block is processed)*
Starting from the initial state, after $1 + 16 + 320 + 3 + 2 = 342$ *clock cycles, if the number of blocks to be processed is higher than 0, then the system is again in the initial state, otherwise it is in the idle state. In both cases RAM is modified and a, b, c, d, e hold the result of the digest for one block.*

The theorem above is obtained by combining Theorems 3 to 7.

Theorem 9. *(VHDL design vs. Intermediate digest function)*
Starting from any state, for any message of nb_blocks stored at base_address in RAM, after the execution of Sha_Vhdl for $3 + 342 * nb\_blocks$ *clock cycles, the system is in its final state (done* $= 1$*) and the values of the output are equal to the result of the Digest-Impl applied to the message.*

$\forall$ *St = (LOCALS,RAM),*
(New-LOCALS, New-RAM) = Sha_Vhdl (First ($3 + 342 * nb\_blocks$*, Input), St)*
Implies
New-LOCALS.done $= 1 \wedge$ *New-LOCALS.state = idle* $\wedge$
ABC_Vars(New-LOCALS) =
Digest-Impl (Initial_Hash_Val, RAM, base_address, nb_blocks, nb_blocks)
$\wedge \forall$ *others elem* $\in$ *LOCALS, New-LOCALS.elem = Initialized.*

First we write a more general form of the theorem by replacing *nb_blocks* with *blocks_left* and stating that starting from the *init* state, after $342 * blocks\_left$ clock cycles, the design processed *blocks_left* blocks.

The generalized theorem is proved by induction using the general scheme defined before, with $step = 342$ and $j = Bv\text{-}nat\text{-}be\ (blocks\_left)$. Then, by combining it with Theorem 1 and Theorem 2 we prove Theorem 9.

4.3 Intermediate Functions vs Specification

Until now we proved that *Digest-Impl* models correctly the behavior of Sha_Vhdl. The next step is to prove the equivalence between the intermediate digest function and the digest specification function *Digest-Spec* for the same message.

Theorem 10. *(Intermediate digest vs. Specification digest function)*
Digest-Impl (Initial_Hash_Val, RAM, base_address, nb_blocks, nb_blocks) =
Digest-Spec(Parsing (Get-Message-from-RAM (nb_blocks, base_address, RAM),512), Initial_Hash_Val)

Get-Message-from-RAM takes *nb_blocks* of 16 32-bit words stored in the RAM starting at *Address* and concatenates them.

We first prove a generalized form of the property: the implementation model and the specification model computes the same message digest for same arbitrary initial hash values, and for $k = nb\_block - blocks\_left$ blocks of message.

The proof uses the induction scheme generated by *Digest-Impl* and a large number of lemmas. Only the top level ones are briefly described:

Lemma. After processing k blocks, the memory storing the rest of the message to be processed (starting from the address $base\_address + 16 * k$) is unaltered.

Lemma. The result of the computation of the digest of one block is the same in both specification and the implementation model.
Digest-one-block-Impl
(80, '00000000', ABC_Vars, RAM, base_address, nb_blocks, blocks_left) =
Digest-one-block-Spec (0, ABC_Vars, (*Get-Block-from-RAM* (Address, RAM)))

For the current block, Address = $base\_address + 16 * (nb\_block - blocks\_left)$.

The proof also uses several properties:

- The computation of the word $W_{(count\ mod\ 16)}$ is the same in both the specification and the implementation model.
- The left rotating operation and the logical function F used in the implementation are equal with the specification defined ones.
- In both models the replacement of the word $W_{(count\ mod\ 16)}$ with the new computed one has the same effect on the initial message.

4.4 The Correctness Theorem

Using Theorems 9 and 10 we prove that Sha_Vhdl implements correctly *Digest_Spec*:

Theorem 11. *Starting from any state, for any message of nb blocks, stored in RAM at address base_address, after the execution of* Sha_Vhdl *for* $3 + 342 * nb$

clock cycles, the system is in its final state and the values of the output are equal to the result of Digest-Spec on the same message.

$$
\begin{aligned}
&\forall\ St = (LOCALS,\ RAM),\\
&(New\text{-}LOCALS,\ New\text{-}RAM) = Sha\_Vhdl\ (First\ (3{+}342{*}nb,\ List\_Inputs),\ St)\\
&Implies\\
&\quad New\text{-}LOCALS.done = 1 \wedge New\text{-}LOCALS.state = idle \wedge\\
&\quad ABC\_Vars(New\text{-}LOCALS) =\\
&\qquad Digest\text{-}Spec\ (Parsing\ (Get\text{-}Message\text{-}from\text{-}RAM\ (nb\_blocks,\ Address,\ RAM),\\
&\qquad\qquad 512),\ Initial\_Hash\_Val)
\end{aligned}
$$

All theorems use a large number of properties that we proved about bit-vectors and operations with bit vectors (logical, arithmetic, concatenation, shifting, conversions, etc), about the RAM and *List_inputs*. The overall proof, including the two models, needed 150 functions and 750 theorems, from which 45% are reusable.

5 Conclusion

The benefits of our research for the project are multiple. For proving a SHA - 1 circuit, we produced executable and reusable specifications for a standard algorithm that was previously available under an informal notation only and we also developed a stepwise method for the proof of the implementation vs specification, based on the control automaton of the RTL. Our current work systematically reuses this approach on a library of components (AES, TDES) for secure circuits.

The extraction of a finite state machine (FSM) from a logic-level RTL is at the basis of all sequential verification tools [7]. The implementation verification of high-level data oriented circuits against a more abstract specification was successfull treated with tools like Uclid [8], TLSim [9]. Yet, from the available publications, these tools require that both the implementation and the specification be modeled in their specific and very restricted HDL. They also rely on uninterpreted functions and a fixed (and small) number of implementation cycles to compute the specified result. In contrast, we allow a very general algorithmic specification, and reason on the results of an efficient automatic FSM extraction from a VHDL design (not necessarily logic-level). Moreover, inductive reasoning is needed to prove results over finite but unbounded input streams. This is precisely where our work takes all its significance.

A couple of errors were also uncovered in the initial VHDL, the most serious being an excessive number of cycles in the digest computation and illegal writing operations on the memory. The use of an executable logic was key to the successful validation of both the specification and the RTL, as it provides an easy model debugging facility.

We believe that the design of a reusable core module should increasingly come with its formal proof of correctness: our work demonstrates that this is feasible, and shows a strategy to reach this goal.

Acknowledgments. We would like to thank our ISIA2 project partners for providing the VHDL implementation of the SHA-1 core and for helping us understand the design.

References

1. National Institute of Standards and Technology: "Secure Hash Standard", Federal Information Processing Standards Publication 180-2, 2002.
2. D. Toma, D. Borrione: "SHA Formalization", ACL2 Workshop, Boulder, USA, 2003.
3. D. Toma, D. Borrione: "Verification of a cryptographic circuit: SHA-1 using ACL2", ACL2 Workshop, Austin, USA, 2004.
4. M. Kaufmann, P. Manolios, and J S. Moore: "Computer-Aided reasoning: ACL2 An approach."(Vol.1) and "ACL2 Case Studies" (Vol.2), Kluwer Academic Press, 2000.
5. D. Toma, D. Borrione, G. Al-Sammane: "Combining several paradigms for circuit validation and verification", in Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, LNCS, Vol 3362/2005, pp. 229.
6. IEEE CS: 1076.6-1999 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, New York; March 2000.
7. O. Coudert, C. Berthet, J.C. Madre: "Verification of synchronous sequential machines based on symbolic execution", in Automatic Verification Methods for Finite State Systems, LNCS No407, Springer, pp.365-373.
8. R. E. Bryant, S. K. Lahiri, S. A. Seshia: "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions", in Computer Aided Verification, LNCS 2404, pp. 78-99, 2002.
9. M. N. Velev, R. E. Bryant: "EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations", in Computer Aided Verification, LNCS 2102, 2001.

From PSL to LTL: A Formal Validation in HOL

Thomas Tuerk and Klaus Schneider

Reactive Systems Group, Department of Computer Science,
University of Kaiserslautern, P.O. Box 3049,
67653 Kaiserslautern, Germany
{tuerk, klaus.schneider}@informatik.uni-kl.de
http://rsg.informatik.uni-kl.de

Abstract. Using the HOL theorem prover, we proved the correctness of a translation from a subset of Accellera's property specification language PSL to linear temporal logic LTL. Moreover, we extended the temporal logic hierarchy of LTL that distinguishes between safety, liveness, and more difficult properties to PSL. The combination of the translation from PSL to LTL with already available translations from LTL to corresponding classes of ω-automata yields an efficient translation from PSL to ω-automata. In particular, this translation generates liveness or safety automata for corresponding PSL fragments, which is important for several applications like bounded model checking.

1 Introduction

Model checking and equivalence checking are state of the art in hardware circuit design flows. Standardised languages like the hardware description languages VHDL [2,30] and Verilog [21] are widespread and allow the convenient exchange of modules, which can also be sold as IP blocks. However, specifications of temporal properties that are required for model checking cannot be easily described with these languages [23,6]. Hence, the research on model checking during the last two decades considered mainly temporal logics like LTL [22], CTL [9] and CTL* [10], and other formalisms like ω-automata [29], monadic second order logics, and the μ-calculus [25].

The discussed temporal logics differ dramatically in terms of syntax, semantics, expressiveness and the complexity of the related verification problem. For example, LTL model checking is PSPACE-complete, while CTL model checking can be done in polynomial time. Of course, this corresponds to the different expressive powers of these logics: It can be shown that temporal logics, ω-automata, monadic predicate logics, and the μ-calculus form a hierarchy in terms of expressiveness [25].

The incompatibility of temporal logics used for specification complicates the exchange of data between different tools; which is a situation similar to circuit design before the standardisation of hardware description languages. Hence, the increased industrial interest in verification naturally lead to standardisation efforts for specification logics [5,4]. Accellera's Property Specification Language (PSL) [1] is a result.

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 342–357, 2005.

However, the translation from (the linear time fragment of) PSL to equivalent ω-automata, as required for model-checking, turned out to be quite difficult, although many partial results of this translation already exist: It is well known how LTL can be translated to equivalent ω-automata [33,8,13,28,12]. There is a hierarchy of ω-automata [18,17,31,25] that distinguishes between safety, liveness and four other classes of increasing expressiveness. Recently, the related subsets of LTL of this hierarchy have been syntactically characterised [24,25] and linear-time translations have been presented [24,25] that translate the temporal logic classes to corresponding symbolic descriptions of ω-automata (these can be directly used for symbolic model checking).

In addition to the temporal operators of LTL, PSL also provides certain abort operators whose semantics turned out to be problematic: In [3], a logic RLTL was introduced that extends LTL by an abort operator in order to show the impact of different abort operators on the complexity of the translation and verification. As a result, it turned out that in the worst case, the previous version of PSL lead to a non-elementary blow-up in the translation to ω-automata. For this reason, the semantics of PSL's reset operator has been changed in version 1.1 following the ideas in [3]. Thus, it is not surprising that a significant subset of PSL can now be translated to RLTL. A further translation from RLTL to LTL has already been presented in [3].

The subtle differences of the reset operators in PSL version 1.01 and version 1.1 demonstrate that the semantics of complex temporal logics like PSL should not be underestimated. In fact, PSL is a complex language that includes many special cases. Therefore, we feel the need to formally verify all parts of the translation of PSL to ω-automata by a theorem prover like HOL. To this end, we implemented deep embeddings of RLTL, LTL and ω-automata in HOL[1]. Using the existing deep embedding[2] of PSL [14] and the existing LTL library [26], *we have formally proved the correctness of the entire translation from* PSL *to* ω*-automata via* RLTL *and* LTL. By a detailed examination of the translation from PSL to LTL, we could moreover extend the known temporal logic classes of LTL to corresponding classes of PSL. In particular, we will present in this paper a syntactic characterisation of subsets of PSL in the spirit of [24] that match with corresponding ω-automata classes for safety, liveness and other properties. Translations to safety or liveness automata are of particular interest for bounded model checking as shown in [27].

The paper is organised as follows: In the next section, we present the temporal logics PSL, RLTL and LTL in detail. Then, we briefly sketch the translations from PSL to RLTL and from RLTL to LTL. In Section 5, we then define classes of PSL that correspond with the temporal logic hierarchy [18,24,25] and hence, also with the ω-automaton hierarchy [17,31,25]. Finally, we draw some conclusions and show directions for future work.

[1] The HOL library is available at `http://rsg.informatik.uni-kl.de/tools`.

[2] Although some members of the Accellera Formal Property Language Technical Committee reviewed this embedding, we found a small, until then unknown bug in the embedding.

2 Basics

Temporal logics like LTL, RLTL and PSL use propositional logic to describe (static) properties of the current point of time. The semantics of dynamic, i. e., temporal properties is based on a sequence of points of time, a so-called *path*. Thus, we first define propositional logic and paths in this section.

Definition 1 (Propositional Logic). *Let $\mathcal{V}$ be a set of variables. Then, the* set of propositional formulas over $\mathcal{V}$ *(short* $\mathsf{prop}_{\mathcal{V}}$*) is recursively given as follows:*

- *each variable $v \in \mathcal{V}$ is a propositional formula*
- $\neg\varphi \in \mathsf{prop}_{\mathcal{V}}$*, if* $\varphi \in \mathsf{prop}_{\mathcal{V}}$
- $\varphi \wedge \psi \in \mathsf{prop}_{\mathcal{V}}$*, if* $\varphi, \psi \in \mathsf{prop}_{\mathcal{V}}$

An assignment *over $\mathcal{V}$ is a subset of $\mathcal{V}$. In our context, assignments are also called* states*. The set of all states over $\mathcal{V}$, which is the power set of $\mathcal{V}$, is denoted by $\mathcal{P}(\mathcal{V})$. The semantics of a propositional formula with respect to a state s is given by the relation $\models_{\mathsf{prop}}$ that is defined as follows:*

- $s \models_{\mathsf{prop}} v$ *iff* $v \in s$
- $s \models_{\mathsf{prop}} \neg\varphi$ *iff* $s \not\models_{\mathsf{prop}} \varphi$
- $s \models_{\mathsf{prop}} \varphi \wedge \psi$ *iff* $s \models_{\mathsf{prop}} \varphi$ *and* $s \models_{\mathsf{prop}} \psi$

If $s \models_{\mathsf{prop}} \varphi$ holds, then the assignment s is said to satisfy *the propositional formula φ.*

Moreover, we use the following abbreviations as syntactic sugar:

- $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \rightarrow \psi := \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi := (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
- $\mathsf{true} := v \vee \neg v$ for an arbitrary variable $v \in \mathcal{V}$
- $\mathsf{false} := \neg\mathsf{true}$

A finite word v over a set Σ of length $|v| = n+1$ is a function $v : \{0, \ldots n\} \rightarrow \Sigma$. An infinite word v over Σ is a function $v : \mathbb{N} \rightarrow \Sigma$ and its length is denoted by $|v| = \infty$. The set Σ is called the *alphabet* and the elements of Σ are called *letters*. The finite word of length 0 is called the *empty word* (denoted by ε). For reasons of simplicity, $v(i)$ is often denoted by v^i for $i \in \mathbb{N}$. Using this notation, words are often given in the form $v^0 v^1 v^2 \ldots v^n$ or $v^0 v^1 \ldots$. The set of all finite and infinite words over Σ is denoted by Σ^* and Σ^ω, respectively.

Counting of letters starts with zero, i. e. v^{i-1} refers to the i-th letter of v. Furthermore, $v^{i\cdot\cdot}$ denotes the suffix of v starting at position i, i. e. $v^{i\cdot\cdot} = v^i v^{i+1} \ldots$ for all $i < |v|$. The finite word $v^i v^{i+1} \ldots v^j$ is denoted by $v^{i\cdot\cdot j}$. Notice that in case $j < i$ the expression $v^{i\cdot\cdot j}$ evaluates to the empty word ε. For two words v_1, v_2 with $v_1 \in \Sigma^*$, we write $v_1 v_2$ for their concatenation. Finally, we write l^ω for the infinite word v with $v^j = l$ for all j.

A *path* of a labelled transition system corresponds to a word whose letters are the labels of the states of the path. However, the terminology of PSL does not distinguish between paths and words [1]. Therefore, the terms 'path' and 'word' are also used synonymously in this work.

2.1 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) has been introduced by Pnueli in [22]. LTL essentially consists of propositional logic enriched with the temporal operators X and $\underline{\mathsf{U}}$. The formula $\mathsf{X}\varphi$ means that the property φ holds at the next point of time, $\varphi\ \underline{\mathsf{U}}\ \psi$ means that φ holds until ψ holds and that ψ eventually holds. The operators $\overleftarrow{\underline{\mathsf{X}}}$ and $\overleftarrow{\underline{\mathsf{U}}}$ express the same properties for the past instead of the future. Therefore, the operators X and $\underline{\mathsf{U}}$ are called *future operators*, while $\overleftarrow{\underline{\mathsf{X}}}$ and $\overleftarrow{\underline{\mathsf{U}}}$ are called *past operators*.

LTL without past operators is as expressive as LTL with past operators [11]. For this reason, past operators are often neglected, although new results advocate the use of past operators [19] since some properties require an exponential blow-up when past operators are eliminated. Hence, our deep embedding of LTL contains past operators as well, so that we distinguish between the full logic LTL and its future fragment FutureLTL.

Definition 2 (Syntax of Linear Temporal Logic (LTL)). *The set* $\mathsf{ltl}_\mathcal{V}$ *of* LTL *formulas over a given set of variables* $\mathcal{V}$ *is defined as follows:*

- $p \in \mathsf{ltl}_\mathcal{V}$ *for all* $p \in \mathsf{prop}_\mathcal{V}$
- $\neg\varphi,\ \varphi \wedge \psi \in \mathsf{ltl}_\mathcal{V}$, *if* $\varphi, \psi \in \mathsf{ltl}_\mathcal{V}$
- $\mathsf{X}\varphi,\ \varphi\ \underline{\mathsf{U}}\ \psi \in \mathsf{ltl}_\mathcal{V}$, *if* $\varphi, \psi \in \mathsf{ltl}_\mathcal{V}$
- $\overleftarrow{\underline{\mathsf{X}}}\varphi,\ \varphi\ \overleftarrow{\underline{\mathsf{U}}}\ \psi \in \mathsf{ltl}_\mathcal{V}$, *if* $\varphi, \psi \in \mathsf{ltl}_\mathcal{V}$

As usual a lot of further temporal operators can be defined as syntactic sugar like $\mathsf{F}\varphi := (\mathsf{true}\ \underline{\mathsf{U}}\ \psi)$, $\mathsf{G}\varphi := \neg\mathsf{F}\neg\varphi$, $\varphi\ \mathsf{U}\ \psi := \varphi\ \underline{\mathsf{U}}\ \psi \vee \mathsf{G}\varphi$, and $\varphi\ \mathsf{B}\ \psi := \neg(\neg\varphi)\ \underline{\mathsf{U}}\ \psi$. LTL with the operators $\underline{\mathsf{U}}$ and X is, however, already expressively complete with respect to the first order theory of linear orders [25].

Definition 3 (Semantics of Linear Temporal Logic (LTL)). *For* $b \in \mathsf{prop}_\mathcal{V}$ *and* $\varphi, \psi \in \mathsf{ltl}_\mathcal{V}$ *the semantics of* LTL *with respect to an infinite word* $v \in \mathcal{P}(\mathcal{V})^\omega$ *and a point of time* $t \in \mathbb{N}$ *is given as follows:*

- $v \models^t_{\mathsf{ltl}} b$ *iff* $v^t \models_{\mathsf{prop}} b$
- $v \models^t_{\mathsf{ltl}} \neg\varphi$ *iff* $v \not\models^t_{\mathsf{ltl}} \varphi$
- $v \models^t_{\mathsf{ltl}} \varphi \wedge \psi$ *iff* $v \models^t_{\mathsf{ltl}} \varphi$ *and* $v \models^t_{\mathsf{ltl}} \psi$
- $v \models^t_{\mathsf{ltl}} \mathsf{X}\varphi$ *iff* $v \models^{t+1}_{\mathsf{ltl}} \varphi$
- $v \models^t_{\mathsf{ltl}} \varphi\ \underline{\mathsf{U}}\ \psi$ *iff* $\exists k.\ k \geq t\ \wedge\ v \models^k_{\mathsf{ltl}} \psi\ \wedge\ \forall j.\ t \leq j < k \rightarrow v \models^j_{\mathsf{ltl}} \varphi$
- $v \models^t_{\mathsf{ltl}} \overleftarrow{\underline{\mathsf{X}}}\varphi$ *iff* $t > 0\ \wedge\ v \models^{t-1}_{\mathsf{ltl}} \varphi$
- $v \models^t_{\mathsf{ltl}} \varphi\ \overleftarrow{\underline{\mathsf{U}}}\ \psi$ *iff* $\exists k.\ k \leq t\ \wedge\ v \models^k_{\mathsf{ltl}} \psi\ \wedge\ \forall j.\ k < j \leq t \rightarrow v \models^j_{\mathsf{ltl}} \varphi$

A word $v \in \mathcal{P}(\mathcal{V})^\omega$ satisfies *a* LTL *formula* $\varphi \in \mathsf{ltl}_\mathcal{V}$ *(written as* $v \models_{\mathsf{ltl}} \varphi$*) iff* $v \models^0_{\mathsf{ltl}} \varphi$.

2.2 Reset Linear Temporal Logic (RLTL)

To evaluate a formula $\varphi\ \underline{\mathsf{U}}\ \psi$, one has to consider a (potentially infinite) prefix of a path, namely the prefix up to a state where $\neg(\varphi \wedge \neg\psi)$ holds. As simulations may stop before that prefix is completely examined, the evaluation of formulas could be incomplete, and is thus aborted. In order to return a definite truth value, abort operators are introduced. In particular, RLTL [3] is such an extension of FutureLTL:

Definition 4 (Syntax of Reset Linear Temporal Logic (RLTL)). *The following mutually recursive definitions introduce the set* $\mathsf{rltl}_\mathcal{V}$ *of* RLTL *formulas over a given set of variables* $\mathcal{V}$*:*

- *each propositional formula* $p \in \mathsf{prop}_\mathcal{V}$ *is a* RLTL *formula*
- $\neg\varphi$, $\varphi \wedge \psi \in \mathsf{rltl}_\mathcal{V}$, *if* $\varphi, \psi \in \mathsf{rltl}_\mathcal{V}$
- $\mathsf{X}\varphi$, $\varphi \,\underline{\mathsf{U}}\, \psi \in \mathsf{rltl}_\mathcal{V}$, *if* $\varphi, \psi \in \mathsf{rltl}_\mathcal{V}$
- $\mathsf{ACCEPT}(\varphi, b) \in \mathsf{rltl}_\mathcal{V}$, *if* $\varphi \in \mathsf{rltl}_\mathcal{V}$, $b \in \mathsf{prop}_\mathcal{V}$

Some operators like $\neg, \wedge$ or $\underline{\mathsf{U}}$ are used by several logics discussed in this paper. In most cases, it is clear by the context or it does not matter to which logic one of these operators belongs. If it matters, we use subscripts like $\neg_{\mathsf{prop}}$, $\neg_{\mathsf{ltl}}$ and $\neg_{\mathsf{rltl}}$. For example, with $a, b, c \in \mathcal{V}$, note that $\langle \{a,b\}\emptyset^\omega, a, b\rangle \models^0_{\mathsf{rltl}} \neg_{\mathsf{prop}} c$ holds, but $\langle \{a,b\}\emptyset^\omega, a, b\rangle \models^0_{\mathsf{rltl}} \neg_{\mathsf{rltl}} c$ does not hold.

Definition 5 (Semantics of Reset Linear Temporal Logic (RLTL)). *The semantics of* LTL *is defined with respect to a word* v *and a point of time* t*. To define the semantics of* RLTL*, an* acceptance condition $a \in \mathsf{prop}_\mathcal{V}$ *and a* rejection condition $r \in \mathsf{prop}_\mathcal{V}$ *are additionally needed. These conditions are used to capture the required information about* ACCEPT *operators in the context of the formula. Thus, for* $b \in \mathsf{prop}_\mathcal{V}$ *and* $\varphi, \psi \in \mathsf{rltl}_\mathcal{V}$*, the semantics of* RLTL *with respect to an infinite word* $v \in \mathcal{P}(\mathcal{V})^\omega$*, acceptance / rejection conditions* $a, r \in \mathsf{prop}_\mathcal{V}$ *and a point of time* $t \in \mathbb{N}$ *is defined as follows:*

- $\langle v, a, r\rangle \models^t_{\mathsf{rltl}} b$ *iff* $v^t \models_{\mathsf{prop}} a$ *or* $(v^t \models_{\mathsf{prop}} b$ *and* $v^t \not\models_{\mathsf{prop}} r)$
- $\langle v, a, r\rangle \models^t_{\mathsf{rltl}} \neg\varphi$ *iff* $\langle v, r, a\rangle \not\models^t_{\mathsf{rltl}} \varphi$
- $\langle v, a, r\rangle \models^t_{\mathsf{rltl}} \varphi \wedge \psi$ *iff* $\langle v, a, r\rangle \models^t_{\mathsf{rltl}} \varphi$ *and* $\langle v, a, r\rangle \models^t_{\mathsf{rltl}} \psi$
- $\langle v, a, r\rangle \models^t_{\mathsf{rltl}} \mathsf{X}\varphi$ *iff* $v^t \models_{\mathsf{prop}} a$ *or* $\left(\langle v, a, r\rangle \models^{t+1}_{\mathsf{rltl}} \varphi \text{ and } v^t \not\models_{\mathsf{prop}} r\right)$
- $\langle v, a, r\rangle \models^t_{\mathsf{rltl}} \varphi \,\underline{\mathsf{U}}\, \psi$
  $$\text{iff } \exists k.\ k \geq t \ \wedge\ \langle v, a, r\rangle \models^k_{\mathsf{rltl}} \psi \ \wedge\ \forall j.\ t \leq j < k \rightarrow \langle v, a, r\rangle \models^j_{\mathsf{rltl}} \varphi$$
- $\langle v, a, r\rangle \models^t_{\mathsf{rltl}} \mathsf{ACCEPT}(\varphi, b)$ *iff* $\langle v, a \vee (b \wedge \neg r), r\rangle \models^t_{\mathsf{rltl}} \varphi$

A word $v \in \mathcal{P}(\mathcal{V})^\omega$ *satisfies a* RLTL *formula* $\varphi \in \mathsf{rltl}_\mathcal{V}$ *(written as* $v \models_{\mathsf{rltl}} \varphi$*) iff* $\langle v, \mathsf{false}, \mathsf{false}\rangle \models^0_{\mathsf{rltl}} \varphi$ *holds.*

$\mathsf{ACCEPT}(\varphi, b)$ aborts the evaluation of the formula φ as soon as the propositional condition b holds: assume we have to check $v \models_{\mathsf{rltl}} \mathsf{ACCEPT}(\varphi \,\underline{\mathsf{U}}\, \psi, b)$ with propositional formulas φ, ψ and b, but only know a finite prefix of v, say $v^{0..t}$. Assume further that on every state v^i with $i \leq t$, we have $v^i \models_{\mathsf{prop}} \varphi \wedge \neg\psi$. Then, we can not decide whether $v \models_{\mathsf{ltl}} \varphi \,\underline{\mathsf{U}}\, \psi$ holds, but nevertheless $v \models_{\mathsf{rltl}} \mathsf{ACCEPT}((\varphi \,\underline{\mathsf{U}}\, \psi), b)$ holds, provided that $v^t \models_{\mathsf{prop}} b$ holds.

For example, the word $\{a\}\{c\}\emptyset^\omega$ does not satisfy the RLTL formula $a \,\underline{\mathsf{U}}\, b$ (since b is never satisfied), but it satisfies the formula $\mathsf{ACCEPT}(a \,\underline{\mathsf{U}}\, b, c)$, since c aborts the until then incomplete evaluation of $a \,\underline{\mathsf{U}}\, b$. On the other hand, the word $\emptyset\{c\}\emptyset^\omega$ does not satisfy $\mathsf{ACCEPT}(a \,\underline{\mathsf{U}}\, b, c)$, since the evaluation of $a \,\underline{\mathsf{U}}\, b$ is completed before c occurs. To understand the impact of the acceptance and rejection conditions and thus, to understand the semantics of the ACCEPT operator, the following lemma is important:

Lemma 1 (Immediate Accept or Reject). *For all infinite words $v \in \mathcal{P}(\mathcal{V})^\omega$, all formulas $\varphi \in \mathsf{rltl}_\mathcal{V}$, all acceptance / rejection conditions $a, r \in \mathsf{prop}_\mathcal{V}$ and all points of time $t \in \mathbb{N}$, the following holds:*

$$\begin{array}{l}\left(v^t \models_{\mathsf{prop}} a \ \wedge \ v^t \not\models_{\mathsf{prop}} r\right) \Longrightarrow \langle v, a, r\rangle \models^t_{\mathsf{rltl}} \varphi \text{ and} \\ \left(v^t \not\models_{\mathsf{prop}} a \ \wedge \ v^t \models_{\mathsf{prop}} r\right) \Longrightarrow \langle v, a, r\rangle \not\models^t_{\mathsf{rltl}} \varphi\end{array}$$

The lemma can be easily proved by structural induction[3] and states that if the acceptance condition immediately holds, every formula (even false) is true. On the other hand, if the rejection condition holds, every formula (even true) is false.

If the acceptance and the rejection condition would hold at the same point of time, then a lot of problems would occur with the semantics. Fortunately, all pairs of acceptance / rejection conditions (a, r) that appear during the evaluation of RLTL formulas satisfy the invariant $\forall s.\, s \models_{\mathsf{prop}} \neg(a \wedge r)$: Initially, the pair $(\mathsf{false}, \mathsf{false})$ is used, and the rules that determine the semantics are easily seen to maintain the invariant.

Therefore, $\forall s.\, s \models_{\mathsf{prop}} \neg(a \wedge r)$ can be assumed for pairs of acceptance / rejection conditions (a, r). This assumption simplifies some proofs, because unreasonable cases can be excluded. In particular, it does not matter if $\neg_{\mathsf{prop}}$ or $\neg_{\mathsf{rltl}}$ is used, if $\forall s.\, s \models_{\mathsf{prop}} \neg(a \wedge r)$ holds[4]. Moreover, the invariant $\neg(a \wedge r)$ is necessary to formulate certain important lemmata like the following one:

Lemma 2. *For all infinite words $v_1, v_2 \in \mathcal{P}(\mathcal{V})^\omega$, all formulas $\varphi \in \mathsf{rltl}_\mathcal{V}$, all acceptance / rejection conditions $a, r \in \mathsf{prop}_\mathcal{V}$ and all points of time $t \in \mathbb{N}$, the following holds[5]:*

$$\begin{array}{l}\Big(\exists k.\, k \geq t \ \wedge \ v_1^{t..k-1} = v_2^{t..k-1} \ \wedge \\ \quad \Big(\left(v_1^k \models_{\mathsf{prop}} a \ \wedge \ v_2^k \models_{\mathsf{prop}} a \ \wedge \ v_1^k \not\models_{\mathsf{prop}} r \ \wedge \ v_2^k \not\models_{\mathsf{prop}} r\right) \vee \\ \quad\quad \left(v_1^k \not\models_{\mathsf{prop}} a \ \wedge \ v_2^k \not\models_{\mathsf{prop}} a \ \wedge \ v_1^k \models_{\mathsf{prop}} r \ \wedge \ v_2^k \models_{\mathsf{prop}} r\right)\Big)\Big) \Longrightarrow \\ \Big(\langle v_1, a, r\rangle \models^t_{\mathsf{rltl}} \varphi \Longleftrightarrow \langle v_2, a, r\rangle \models^t_{\mathsf{rltl}} \varphi\Big)\end{array}$$

Lemma 2 states that if either the acceptance or the rejection condition holds at some point of time $k \geq t$, then it is sufficient to consider the finite prefix $v^{t..k}$ to evaluate arbitrary RLTL formulas at position t. This does no longer hold if both the acceptance and the rejection condition would hold at some point of time: For example, we have $\langle\{a, b\}\emptyset^\omega, a, b\rangle \models^0_{\mathsf{rltl}} a \ \underline{\mathsf{U}} \ \neg_{\mathsf{rltl}} c$, but $\langle\{a, b\}\{c\}\emptyset^\omega, a, b\rangle \not\models^0_{\mathsf{rltl}} a \ \underline{\mathsf{U}} \ \neg_{\mathsf{rltl}} c$. The remaining RLTL operators have the same semantics as the corresponding LTL operators (since RLTL is a superset of LTL).

2.3 Accellera's Property Specification Language

As mentioned above, PSL is a standardised industrial-strength property specification language [1]. PSL was chartered by the Functional Verification Technical

[3] Theorem `RLTL_ACCEPT_REJECT_THM` in theory `ResetLTL_Lemmata`.
[4] Theorem `RLTL_SEM_PROP_RLTL_OPERATOR_EQUIV` in theory `ResetLTL`.
[5] Theorem `RLTL_EQUIV_PATH_STRONG_THM` in theory `ResetLTL_Lemmata`.

Committee of Accellera. The Sugar language [5] was chosen as the basis for PSL. The Language Reference Manual for PSL version 1.0 was released in April 2003. Finally, in June 2004 version 1.1 [1] was released, where some anomalies (like those reported in [3]) were corrected.

PSL is designed as an input language for formal verification and simulation tools as well as a language for documentation. Therefore, it has to be easy to read, and at the same time, it must be precise and highly expressive. In particular, PSL contains features for simulation like finite paths, features for hardware specification like clocked statements and a lot of syntactic sugar.

PSL consists of four layers: The Boolean layer, the temporal layer, the verification layer and the modelling layer. The *Boolean layer* is used to construct expressions that can be evaluated in a single state. The *temporal layer* is the heart of the language. It is used to express properties concerning more than one state, i. e. temporal properties. The temporal layer is divided into the *Foundation Language* (FL) and the *Optional Branching Extension* (OBE). FL is, like LTL, a linear time temporal logic. In contrast, OBE is essentially the branching time temporal logic CTL [9], which is widely used and well understood. The *verification layer* has the task to instruct tools to perform certain actions on the properties expressed by the temporal layer. Finally, the *modelling layer* is used to describe assumptions about the behaviour of inputs and to model auxiliary hardware that is not part of the design. Additionally, PSL comes in four flavours, corresponding to the hardware description languages SystemVerilog, Verilog, VHDL and GDL. These flavours provide a syntax for PSL that is similar to the syntax of the corresponding hardware description language.

In this paper, only the Boolean and the temporal layers will be considered. Furthermore, mainly the formal syntax of PSL is used, which differs from the syntax of all four flavours. However, some operators are denoted differently to the formal syntax to avoid confusion with LTL operators that have the same syntax, but a different semantics.

In this paper, only FL is considered. Therefore, only this subset of PSL is formally introduced here. FL is a linear temporal logic that consists of:

- propositional operators
- future temporal (LTL) operators
- a clocking operator for defining the granularity of time, which may vary for subformulas
- Sequential Extended Regular Expressions (SEREs), for defining finite regular patterns, together with strong and weak promotions of SEREs to formulas and an implication operator for predicating a formula on match of the pattern specified by a SERE
- an abort operator

Due to lack of space, only the subset of PSL that is interesting for the translation will be presented. Therefore, clocks and SEREs are omitted in the following.

The definition of the formal semantics of PSL makes use of two special states $\top$ and $\bot$. State $\top$ satisfies every propositional formula, even the formula false,

and state $\bot$ satisfies no propositional formula, even the formula true is not satisfied. Using these two special states, the semantics of a propositional formula $\varphi \in \mathsf{prop}_{\mathcal{V}}$ with respect to a state $s \in \mathcal{P}(\mathcal{V}) \cup \{\top, \bot\}$ is defined as follows:

- $\top \models_{\mathsf{xprop}} \varphi$
- $\bot \not\models_{\mathsf{xprop}} \varphi$
- $s' \models_{\mathsf{xprop}} \varphi$ iff $s' \models_{\mathsf{prop}} \varphi$ for $s' \in \mathcal{P}(\mathcal{V})$, i. e. for $s' \notin \{\top, \bot\}$

For a given set of variables $\mathcal{V}$, the set of *extended states over* $\mathcal{V}$ is denoted by $\mathcal{XP}(\mathcal{V}) := \mathcal{P}(\mathcal{V}) \cup \{\top, \bot\}$. The definition of the formal syntax of PSL uses a special function for words over these extended states. For finite or infinite words $w \in \mathcal{XP}(\mathcal{V})^\omega \cup \mathcal{XP}(\mathcal{V})^*$, the word $\overline{w}$ denotes the word over states that is obtained from w by replacing every $\top$ with $\bot$ and vice versa, i. e. for all $i < |w|$, the following holds:

$$\overline{w}^i := \begin{cases} \bot & : \text{if } w^i = \top \\ \top & : \text{if } w^i = \bot \\ w^i & : \text{otherwise} \end{cases}$$

Using these extended states and words over these states, it is possible to define the formal syntax and semantics of SERE-free, unclocked FL (short SUFL):

Definition 6 (Syntax of Unclocked, SERE-free Foundation Language (SUFL)). *The set of* SUFL*-formulas* $\mathsf{sufl}_{\mathcal{V}}$ *over a given set of variables* $\mathcal{V}$ *is defined as follows:*

- $p, p! \in \mathsf{sufl}_{\mathcal{V}}$, *if* $p \in \mathsf{prop}_{\mathcal{V}}$
- $\neg\varphi \in \mathsf{sufl}_{\mathcal{V}}$, *if* $\varphi \in \mathsf{sufl}_{\mathcal{V}}$
- $\varphi \wedge \psi \in \mathsf{sufl}_{\mathcal{V}}$, *if* $\varphi, \psi \in \mathsf{sufl}_{\mathcal{V}}$
- $\underline{\mathsf{X}}\varphi$, $\varphi\ \underline{\mathsf{U}}\ \psi$[6] $\in \mathsf{sufl}_{\mathcal{V}}$, *if* $\varphi, \psi \in \mathsf{sufl}_{\mathcal{V}}$
- $\varphi\ \mathsf{ABORT}\ b \in \mathsf{sufl}_{\mathcal{V}}$, *if* $\varphi \in \mathsf{sufl}_{\mathcal{V}}$, $b \in \mathsf{prop}_{\mathcal{V}}$

Definition 7 (Semantics of SUFL). *For propositional formulas* $b \in \mathsf{prop}_{\mathcal{V}}$ *and* SUFL *formulas* $\varphi, \psi \in \mathsf{sufl}_{\mathcal{V}}$, *the semantics of unclocked* SUFL *with respect to a finite or infinite word* $v \in \mathcal{XP}(\mathcal{V})^* \cup \mathcal{XP}(\mathcal{V})^\omega$ *is defined as follows:*

- $v \models_{\mathsf{sufl}} b$ *iff* $|v| = 0$ *or* $v^0 \models_{\mathsf{xprop}} b$
- $v \models_{\mathsf{sufl}} b!$ *iff* $|v| > 0$ *and* $v^0 \models_{\mathsf{xprop}} b$
- $v \models_{\mathsf{sufl}} \neg\varphi$ *iff* $\overline{v} \not\models_{\mathsf{sufl}} \varphi$
- $v \models_{\mathsf{sufl}} \varphi \wedge \psi$ *iff* $v \models_{\mathsf{sufl}} \varphi$ *and* $v \models_{\mathsf{sufl}} \psi$
- $v \models_{\mathsf{sufl}} \underline{\mathsf{X}}\varphi$ *iff* $|v| > 1$ *and* $v^{1..} \models_{\mathsf{sufl}} \varphi$
- $v \models_{\mathsf{sufl}} \varphi\ \underline{\mathsf{U}}\ \psi$ *iff* $\exists k.\, k < |v|$ *s.t.* $v^{k..} \models_{\mathsf{sufl}} \psi$ *and* $\forall j < k.\, v^{j..} \models \varphi$
- $v \models_{\mathsf{sufl}} \varphi\ \mathsf{ABORT}\ b$ *iff either* $v \models_{\mathsf{sufl}} \varphi$ *or* $\exists j.j < |v|$ *s.t.* $v^j \models_{\mathsf{sufl}} b$ *and* $v^{0..j-1}\top^\omega \models_{\mathsf{sufl}} \varphi$

A word v *satisfies an unclocked* FL *formula* φ *iff* $v \models_{\mathsf{sufl}} \varphi$ *holds.*

[6] Written as $\varphi\ \mathsf{U}\ \psi$ in [1].

As usual, some syntactic sugar is defined for SUFL:

- $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \rightarrow \psi := \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi := (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
- $\mathsf{X}\varphi := \neg\underline{\mathsf{X}}\neg\varphi$
- $\mathsf{F}\varphi := \mathsf{true}\ \underline{\mathsf{U}}\ \varphi$
- $\mathsf{G}\varphi := \neg\mathsf{F}\neg\varphi$
- $\varphi\ \mathsf{U}\ \psi$[7] $:= \varphi\ \underline{\mathsf{U}}\ \psi\ \vee\ \mathsf{G}\varphi$
- $\varphi\ \mathsf{B}\ \psi$[8] $:= \neg(\neg\varphi\ \underline{\mathsf{U}}\ \psi)$

All SUFL operators correspond to RLTL operators. A difference to RLTL is, that SUFL is able to additionally consider finite paths. Thus, for a propositional formula b, a strong variant $b!$ is introduced that does not hold for the empty word ε, while every propositional formula b holds for the empty word. Analogously, $\underline{\mathsf{X}}$ is introduced as a strong variant of X. The semantics of $\underline{\mathsf{X}}$ requires that a next state exists, while $\mathsf{X}\varphi$ trivially holds if no next state exists. For the remaining temporal operator $\underline{\mathsf{U}}$, a weak variant U is already available in RLTL. Apart from finite paths, the meaning of the FL operators is the same as the meaning of the corresponding RLTL operators. The role of the two special states $\top, \bot$ is played by the acceptance / rejection conditions of RLTL. The proof of this connection between PSL and RLTL is one important part of the translation presented in this work and will be explained in Section 3.

3 From PSL to RLTL

As mentioned above, the temporal layer of PSL consists of FL and OBE. OBE is essentially the well known temporal logic CTL [9]. Since CTL can be directly used for model checking without further translations [7,25], this work only considers FL.

FL with SEREs is strictly more expressive than LTL. For example, it is well known that there is no LTL formula expressing that a proposition φ holds at every even point of time [20,32,25]. However, there is an unclocked FL formula with SEREs expressing this property[9]. As RLTL is as expressive as LTL [3], FL with SEREs cannot be translated to RLTL. Therefore, only SERE-free FL formulas are considered. Moreover, clock-statements can be omitted for reasons of simplicity, because clocked formulas can be easily rewritten to equivalent unclocked ones [1]. Thus, we only consider the translation of unclocked, SERE-free FL to RLTL.

The semantics of SUFL is similar to the semantics of RLTL. There are only two important differences: first, SUFL is able to additionally consider finite paths, and second, SUFL additionally makes use of the special states $\top$ and $\bot$, while RLTL makes use of acceptance and rejection conditions. The first difference is not important in the scope of this paper, because the overall goal is to translate SUFL to ω-automata. Therefore, only infinite paths are of interest. To handle

[7] Written as $[\varphi\ \mathsf{W}\ \psi]$ in [1].
[8] Written as $[\varphi\ \mathsf{BEFORE!\_}\ \psi]$ in [1].
[9] Theorem `PSL_WITH_SERES__STRICTLY_MORE_EXPRESSIVE_THAN__LTL__EXAMPLE` in theory `PSLToRLTL`.

the second difference, the special states $\top$ and $\bot$ are simulated with the acceptance / rejection conditions of RLTL. However, the special states and acceptance / rejection conditions have slightly different semantics: The states $\top$ and $\bot$ determine whether an arbitrary proposition is fulfilled by the current state. However, the remaining states are still important. In contrast, if either the acceptance or the rejection condition occurs, the remaining states can be neglected according to Lemma 2. An example showing this difference is $\bot\{p\}^\omega \models_{\mathsf{sufl}} \underline{\mathsf{X}}p$, but $\langle\{r\}\{p\}^\omega, a, r\rangle \not\models^0_{\mathsf{rltl}} \mathsf{X}p$ for $a, r, p \in \mathcal{V}$. To overcome this slightly different semantics only special inputs are considered:

Definition 8 (PSL-Paths). *A* finite PSL-path *over a set of variables $\mathcal{V}$ is a finite word $v \in \mathcal{P}(\mathcal{V})^*$, i. e. a finite word not containing the states $\top$ and $\bot$. An* infinite PSL-path *over $\mathcal{V}$ is an infinite word $v \in \mathcal{XP}(\mathcal{V})^\omega$ with the following properties:*

- $\forall j.\ v^j{=}\top \longrightarrow v^{j+1}{=}\top$
- $\forall j.\ v^j{=}\bot \longrightarrow v^{j+1}{=}\bot$

The set of all infinite PSL-paths over $\mathcal{V}$ is denoted by $\mathcal{XP}(\mathcal{V})^{\omega^{\top\bot}}$. Notice that $\mathcal{P}(\mathcal{V})^\omega \subset \mathcal{XP}(\mathcal{V})^{\omega^{\top\bot}}$ holds.

In this work, we only consider infinite PSL-paths. At the first glance, this may seem to be a restriction, however, this is not the case: Note that special states are just an auxiliary means used to explain the semantics; however, they do not occur in practice. Hence, only paths that fulfil the additional property of PSL-paths are considered in the following. In [15], PSL-paths are called *proper words*.

Since paths containing the special states $\top$ and $\bot$ are allowed as input of SUFL formulas, but these special states are not allowed as input of RLTL formulas, both paths and formulas have to be translated. To translate the paths, two new atomic propositions t and b are chosen, i. e. t and b do neither occur on the path nor in the formula. Every occurrence of $\top$ on the path is replaced by the state $\{t\}$. In the same way, every occurrence of $\bot$ is replaced by $\{b\}$. For the formula itself, only minor changes are required: Essentially, only the PSL operators are exchanged with the corresponding RLTL operators. Additionally, t and b are used as acceptance and rejection conditions, respectively, while evaluating the translated formula on the translated path.

Lemma 3. *With the definitions of Figure 1, the following are equivalent*[10] *for all $f \in \mathsf{sufl}_\mathcal{V}$, all infinite PSL-paths $v \in \mathcal{XP}(\mathcal{V})^{\omega^{\top\bot}}$ and all $t, b \notin \mathcal{V}$:*

- $v \models_{\mathsf{sufl}} f$
- $\langle \mathsf{RemoveTopBottom}(t, b, v), t, b\rangle \models^0_{\mathsf{rltl}} \mathsf{PSL\_TO\_RLTL}\ f$
- $\mathsf{RemoveTopBottom}(t, b, v) \models_{\mathsf{rltl}} \mathsf{ACCEPT}(\mathsf{REJECT}((\mathsf{PSL\_TO\_RLTL}\ f), b), t)$

Note that t and b never occur at the same point of time on the translated path $\mathsf{RemoveTopBottom}(t, b, v)$.

[10] Theorems `PSL_TO_RLTL_THM` and `PSL_TO_RLTL__ELIM_ACCEPT_REJECT_THM` in theory `PSLToRLTL`.

$$\mathsf{RemoveTopBottom}(t,b,v)^j := \begin{cases} \{t\} & : \textit{if } v^j = \top \\ \{b\} & : \textit{if } v^j = \bot \\ v^j & : \textit{otherwise} \end{cases}$$

function PSL_TO_RLTL(Φ)
  case Φ **of**
    b : **return** b;
    $b!$: **return** b;
    $\neg\varphi$: **return** $\neg$PSL_TO_RLTL(φ);
    $\varphi \wedge \psi$: **return** PSL_TO_RLTL(φ) $\wedge$ PSL_TO_RLTL(ψ);
    $\underline{\mathsf{X}}\varphi$: **return** X(PSL_TO_RLTL(φ));
    $\varphi\ \underline{\mathsf{U}}\ \psi$: **return** PSL_TO_RLTL(φ) $\underline{\mathsf{U}}$ PSL_TO_RLTL(ψ);
    φ ABORT b : **return** ACCEPT(PSL_TO_RLTL(φ), b);
  end
end

Fig. 1. Translation of SUFL to RLTL

The proof of Lemma 3 is based on a structural induction and requires some lemmata about RLTL. In particular, Lemma 1 and 2 are important. To express other important properties in a convenient way, some abbreviations about the occurrence of propositions on a path are convenient:

$$\begin{aligned} \mathsf{NAND\_ON\_PATH}(v,a,r) &:= \forall t.\, \neg(v^t \models_{\mathsf{prop}} a \ \wedge\ v^t \models_{\mathsf{prop}} r) \\ \mathsf{IS\_ON\_PATH}(v,p) &:= \exists t.\, v^t \models_{\mathsf{prop}} p \\ \mathsf{BEFORE\_ON\_PATH}(v,a,b) &:= \forall t.\, (v^t \models_{\mathsf{prop}} b) \Rightarrow \exists t_0.\, (t_0 \leq t \wedge v^{t_0} \models_{\mathsf{prop}} a) \end{aligned}$$

Using these abbreviations, we can formulate the following lemma:

Lemma 4. *For all $v \in \mathcal{P}(\mathcal{V})^\omega$, $a_1, a_2, r \in \mathsf{prop}_\mathcal{V}$, all $\varphi \in \mathsf{rltl}_\mathcal{V}$ and all points of time $t \in \mathbb{N}$, the following holds*[11]*:*

$$\begin{pmatrix} \mathsf{NAND\_ON\_PATH}(v^{t\cdot\cdot}, a_1, r) \ \wedge\ \mathsf{BEFORE\_ON\_PATH}(v^{t\cdot\cdot}, a_1, a_2) \end{pmatrix} \Longrightarrow \\ \begin{pmatrix} \langle v, a_2, r\rangle \models^t_{\mathsf{rltl}} \varphi \Rightarrow \langle v, a_1, r\rangle \models^t_{\mathsf{rltl}} \varphi \end{pmatrix}$$

Informally, this lemma states that valid RLTL formulas do not become invalid if the acceptance condition is strengthened. That is an important property of RLTL. A consequence of Lemma 4 is:

Lemma 5. *For all $v \in \mathcal{P}(\mathcal{V})^\omega$, $a_1, a_2, r \in \mathsf{prop}_\mathcal{V}$, all $\varphi \in \mathsf{rltl}_\mathcal{V}$ and all points of time $t \in \mathbb{N}$, the following property holds*[12]*:*

$$\begin{pmatrix} \mathsf{NAND\_ON\_PATH}(v^{t\cdot\cdot}, a_1, r) \ \wedge\ \mathsf{NAND\_ON\_PATH}(v^{t\cdot\cdot}, a_1, r) \end{pmatrix} \Longrightarrow \\ \begin{pmatrix} \langle v, a_1 \vee a_2, r\rangle \models^t_{\mathsf{rltl}} \varphi \Longleftrightarrow \left(\langle v, a_1, r\rangle \models^t_{\mathsf{rltl}} \varphi \ \vee\ \langle v, a_2, r\rangle \models^t_{\mathsf{rltl}} \varphi \right) \end{pmatrix}$$

[11] Theorem `RLTL_SEM_TIME___ACCEPT_BEFORE_ON_PATH` in theory `ResetLTL_Lemmata`.
[12] Theorem `RLTL_SEM_TIME___ACCEPT_OR_THM` in theory `ResetLTL_Lemmata`.

Using these lemmata about the acceptance / rejection conditions of RLTL, the remaining proof of Lemma 3 by structural induction is mainly technical. The cases for $b, b!, \neg\varphi$ and $\varphi \wedge \psi$ are obvious. The case for $\underline{\mathsf{X}}\varphi$ uses the fact that only infinite PSL-paths are considered, the rest it is technical. The same holds for $\varphi\ \underline{\mathsf{U}}\ \psi$. Therefore, the only interesting case is that for φ ABORT b, where the presented lemmata about RLTL are required in a case analysis.

The usage of HOL to formally prove Lemma 3 has been shown valuable. The case analysis used to prove the case for ABORT is quite tricky. During this case analysis, a small, until then unknown bug in Mike Gordon's deep-embedding of PSL has been discovered: The unclocked semantics of ABORT is defined by $v \models_{\mathsf{sufl}} \varphi$ ABORT b iff either $v \models_{\mathsf{sufl}} \varphi$ or $\exists j.j < |v|$ s.t. $v^j \models_{\mathsf{sufl}} b$ and $v^{0..j-1}\top^\omega \models_{\mathsf{sufl}} \varphi$ holds. This has been literally implemented in HOL. In case of $j = 0$, the word $v^{0..0-1}\top^\omega$ is evaluated to $\top^\omega$ by the formal semantics of PSL. However, because the datatype used to model j in HOL represents natural numbers, $v^{0..0-1}\top^\omega$ evaluated to $v^{0..0}\top^\omega$ and therefore, to $v^0\top^\omega$ in the HOL representation. After reporting this bug to Mike Gordon, it has been fixed.

Lemma 3 is the central result for the translation of PSL to RLTL. It considers arbitrary infinite PSL-paths as inputs. However, one is usually interested only in paths without special states. Restricting the allowed input paths, Lemma 3 directly leads to the following theorem:

Theorem 1 (Translation of SUFL to RLTL). *For all infinite words $v \in \mathcal{P}(\mathcal{V})^\omega$ and all $\varphi \in \mathsf{sufl}_\mathcal{V}$[13], the following holds:*

$$v \models_{\mathsf{sufl}} \varphi \Longleftrightarrow v \models_{\mathsf{rltl}} \mathsf{PSL\_TO\_RLTL}(\varphi).$$

4 From RLTL to LTL

The translation of RLTL to LTL that is used here is due to [3]. The correctness of this translation can be easily proved by structural induction.

Theorem 2 (Translation of RLTL to LTL). *With the definition of Figure 2, the following holds*[14] *for all infinite words $v \in \mathcal{P}(\mathcal{V})^\omega$, all acceptance / rejection conditions $a, r \in \mathsf{prop}_\mathcal{V}$, all RLTL formulas $\varphi \in \mathsf{rltl}_\mathcal{V}$ and all points of time $t \in \mathbb{N}$:*

$$\langle v, a, r\rangle \models^t_{\mathsf{rltl}} \varphi \Longleftrightarrow v \models^t_{\mathsf{ltl}} \mathsf{RLTL\_TO\_LTL}(a, r, \varphi)$$

Obviously, this can be instantiated to:

$$v \models_{\mathsf{rltl}} \varphi \Longleftrightarrow v \models_{\mathsf{ltl}} \mathsf{RLTL\_TO\_LTL}(\mathsf{false}, \mathsf{false}, \varphi)$$

[13] Theorem `PSL_TO_RLTL___NO_TOP_BOT_THM` in theory `PSLToRLTL`.
[14] Theorem `RLTL_TO_LTL_THM` in theory `ResetLTL_Lemmata`.

```
function RLTL_TO_LTL(a, r, Φ)
    case Φ of
        b              : return a ∨ (b ∧ ¬r);
        ¬φ             : return ¬RLTL_TO_LTL(r, a, φ);
        φ ∧ ψ          : return RLTL_TO_LTL(a, r, φ) ∧ RLTL_TO_LTL(a, r, ψ);
        Xφ             : return a ∨ (X(RLTL_TO_LTL(a, r, φ)) ∧ ¬r);
        φ U ψ          : return RLTL_TO_LTL(a, r, φ) U RLTL_TO_LTL(a, r, ψ);
        ACCEPT(φ, b)   : return RLTL_TO_LTL(a ∨ (b ∧ ¬r), r, φ);
    end
end
```

Fig. 2. Translation of RLTL to LTL

5 Temporal Logic Hierarchy for PSL

In [25], LTL classes $\mathsf{LTL_F}$, $\mathsf{LTL_G}$, $\mathsf{LTL_{Prefix}}$, $\mathsf{LTL_{FG}}$, $\mathsf{LTL_{GF}}$ and $\mathsf{LTL_{Streett}}$ are syntactically identified, that are as expressive as deterministic, noncounting liveness ($\mathsf{TDET_F}$), safety ($\mathsf{TDET_G}$), prefix ($\mathsf{TDET_{Prefix}}$), persistence ($\mathsf{TDET_{FG}}$), Büchi ($\mathsf{TDET_{GF}}$) and Streett automata ($\mathsf{TDET_{Streett}}$), respectively.

The translation from PSL to LTL adds additional Boolean expressions to the translated formulas. Adding Boolean expressions does not affect the membership of a formula in these classes. Therefore, it is straightforward to identify classes of unclocked, SERE-free FL, which correspond to the classes of LTL (see Figure 3). Similarly, we have identified a hierarchy of RLTL.

$$
\begin{aligned}
b &\in \mathsf{SUFL_G} \\
b! &\in \mathsf{SUFL_G} \\
\neg\varphi &\in \mathsf{SUFL_G} = \varphi \in \mathsf{SUFL_F} \\
\varphi \wedge \psi &\in \mathsf{SUFL_G} = \varphi \in \mathsf{SUFL_G} \wedge \psi \in \mathsf{SUFL_G} \\
\underline{\mathsf{X}}\varphi &\in \mathsf{SUFL_G} = \varphi \in \mathsf{SUFL_G} \\
\varphi \,\underline{\mathsf{U}}\, \psi &\in \mathsf{SUFL_G} = \mathsf{false} \\
\varphi \;\mathsf{ABORT}\; b &\in \mathsf{SUFL_G} = \varphi \in \mathsf{SUFL_G}
\end{aligned}
\qquad
\begin{aligned}
b &\in \mathsf{SUFL_F} \\
b! &\in \mathsf{SUFL_F} \\
\neg\varphi &\in \mathsf{SUFL_F} = \varphi \in \mathsf{SUFL_G} \\
\varphi \wedge \psi &\in \mathsf{SUFL_F} = \varphi \in \mathsf{SUFL_F} \wedge \psi \in \mathsf{SUFL_F} \\
\underline{\mathsf{X}}\varphi &\in \mathsf{SUFL_F} = \varphi \in \mathsf{SUFL_F} \\
\varphi \,\underline{\mathsf{U}}\, \psi &\in \mathsf{SUFL_F} = \varphi \in \mathsf{SUFL_F} \wedge \psi \in \mathsf{SUFL_F} \\
\varphi \;\mathsf{ABORT}\; b &\in \mathsf{SUFL_F} = \varphi \in \mathsf{SUFL_F}
\end{aligned}
$$

$$
\begin{aligned}
b &\in \mathsf{SUFL_{GF}} \\
b! &\in \mathsf{SUFL_{GF}} \\
\neg\varphi &\in \mathsf{SUFL_{GF}} = \varphi \in \mathsf{SUFL_{FG}} \\
\varphi \wedge \psi &\in \mathsf{SUFL_{GF}} = \varphi \in \mathsf{SUFL_{GF}} \wedge \psi \in \mathsf{SUFL_{GF}} \\
\underline{\mathsf{X}}\varphi &\in \mathsf{SUFL_{GF}} = \varphi \in \mathsf{SUFL_{GF}} \\
\varphi \,\underline{\mathsf{U}}\, \psi &\in \mathsf{SUFL_{GF}} = \varphi \in \mathsf{SUFL_{GF}} \wedge \psi \in \mathsf{SUFL_F} \\
\varphi \;\mathsf{ABORT}\; b &\in \mathsf{SUFL_{GF}} = \varphi \in \mathsf{SUFL_{GF}}
\end{aligned}
\qquad
\begin{aligned}
b &\in \mathsf{SUFL_{FG}} \\
b! &\in \mathsf{SUFL_{FG}} \\
\neg\varphi &\in \mathsf{SUFL_{FG}} = \varphi \in \mathsf{SUFL_{GF}} \\
\varphi \wedge \psi &\in \mathsf{SUFL_{FG}} = \varphi \in \mathsf{SUFL_{FG}} \wedge \psi \in \mathsf{SUFL_{FG}} \\
\underline{\mathsf{X}}\varphi &\in \mathsf{SUFL_{FG}} = \varphi \in \mathsf{SUFL_{FG}} \\
\varphi \,\underline{\mathsf{U}}\, \psi &\in \mathsf{SUFL_{FG}} = \varphi \in \mathsf{SUFL_{FG}} \wedge \psi \in \mathsf{SUFL_{FG}} \\
\varphi \;\mathsf{ABORT}\; b &\in \mathsf{SUFL_{FG}} = \varphi \in \mathsf{SUFL_{FG}}
\end{aligned}
$$

$$
\begin{aligned}
b &\in \mathsf{SUFL_{Prefix}} \\
b! &\in \mathsf{SUFL_{Prefix}} \\
\neg\varphi &\in \mathsf{SUFL_{Prefix}} = \varphi \in \mathsf{SUFL_{Prefix}} \\
\varphi \wedge \psi &\in \mathsf{SUFL_{Prefix}} = \varphi \in \mathsf{SUFL_{Prefix}} \wedge \psi \in \mathsf{SUFL_{Prefix}} \\
\underline{\mathsf{X}}\varphi &\in \mathsf{SUFL_{Prefix}} = \mathsf{X}\varphi \in \mathsf{SUFL_G} \cup \mathsf{SUFL_F} \\
\varphi \,\underline{\mathsf{U}}\, \psi &\in \mathsf{SUFL_{Prefix}} = \varphi \,\underline{\mathsf{U}}\, \psi \in \mathsf{SUFL_G} \cup \mathsf{SUFL_F} \\
\varphi \;\mathsf{ABORT}\; b &\in \mathsf{SUFL_{Prefix}} = \varphi \in \mathsf{SUFL_{Prefix}}
\end{aligned}
\qquad
\begin{aligned}
b &\in \mathsf{SUFL_{Streett}} \\
b! &\in \mathsf{SUFL_{Streett}} \\
\neg\varphi &\in \mathsf{SUFL_{Streett}} = \varphi \in \mathsf{SUFL_{Streett}} \\
\varphi \wedge \psi &\in \mathsf{SUFL_{Streett}} = \varphi \in \mathsf{SUFL_{Streett}} \wedge \psi \in \mathsf{SUFL_{Streett}} \\
\underline{\mathsf{X}}\varphi &\in \mathsf{SUFL_{Streett}} = \mathsf{X}\varphi \in \mathsf{SUFL_{GF}} \cup \mathsf{SUFL_{FG}} \\
\varphi \,\underline{\mathsf{U}}\, \psi &\in \mathsf{SUFL_{Streett}} = \varphi \,\underline{\mathsf{U}}\, \psi \in \mathsf{SUFL_{GF}} \cup \mathsf{SUFL_{FG}} \\
\varphi \;\mathsf{ABORT}\; b &\in \mathsf{SUFL_{Streett}} = \varphi \in \mathsf{SUFL_{Streett}}
\end{aligned}
$$

Fig. 3. Classes of SUFL

We formally proved in HOL that the presented translation translate every PSL class to the corresponding classes of RLTL and LTL. Moreover, the classes of FutureLTL are as expressive as the classes of LTL [25], and FutureLTL is a subset of RLTL and SUFL. Therefore, we formally proved in HOL that the classes of FutureLTL can be translated to the corresponding classes of PSL and RLTL. This leads to the following theorem:

Theorem 3 (Hierarchy of PSL). *For any* $\kappa \in \{\mathsf{G}, \mathsf{F}, \mathsf{Prefix}, \mathsf{GF}, \mathsf{FG}, \mathsf{Streett}\}$*, the logics* LTL_κ*,* $\mathsf{FutureLTL}_\kappa$*,* RLTL_κ *and* SUFL_κ *are as expressive as* TDET_κ*. Furthermore,* LTL*,* FutureLTL*,* RLTL *and* SUFL *are as expressive as* $\mathsf{TDET}_{\mathsf{Streett}}$*.*

6 Conclusion and Future Work

We presented a translation of a significant subset of PSL to LTL. This translation is interesting by its own, since it allows an efficient translation from this significant subset of PSL to ω-automata. Moreover, it is possible to extend the temporal logic hierarchy [18,24,25] to PSL. In particular, we were able to characterise subsets of PSL that can be translated to liveness and safety automata. This is of practical evidence, since these kinds of automata are very useful to handle finite inputs which is required for bounded model checking or for simulation.

Our main goal is to translate PSL to ω-automata. Since the translation of LTL to ω-automata is well known, we have already done a big step. Unfortunately, regular expressions can, in general, not be translated to LTL. However, they can be translated to finite state automata [16]. Therefore, the next step will be to translate PSL directly to ω-automata. We have already deeply embedded automaton formulas [24,25], a symbolic representation of ω-automata. Furthermore, we have validated a basic and an improved translation of LTL to ω-automata, which are both presented in [24,25]. The improved translation allows us to formally validate the translation of SUFL_F, SUFL_G and $\mathsf{SUFL}_{\mathsf{Prefix}}$ to TDET_F, TDET_G or $\mathsf{TDET}_{\mathsf{Prefix}}$, respectively. Next, we will validate more optimised translations. This will allow us to formally validate that also the other classes of PSL can be translated to the corresponding classes of ω-automata. Then, we can use these optimised translations to directly translate a subset of PSL including regular expressions to ω-automata.

References

1. Accellera. Property specification language reference manual, version 1.1. http://www.eda.org, June 2004.
2. ANSI/IEEE Std 1076-1987. *IEEE Standard VHDL Language Reference Manual.* New York, USA, March 1987.
3. Armoni, R., Bustan, D., Kupferman, O., and Vardi, M. Resets vs. aborts in linear temporal logic. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Warsaw, Poland, 2003), H. Garavel and J. Hatcliff, Eds., vol. 2619 of *LNCS*, Springer, pp. 65–80.

4. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M., and Zbar, Y. The ForSpec temporal logic: A new temporal property-specification language. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Grenoble, France, 2002), J.-P. Katoen and P. Stevens, Eds., vol. 2280 of *LNCS*, Springer, pp. 296–211.
5. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., and Rodeh, Y. The temporal logic Sugar. In *Conference on Computer Aided Verification (CAV)* (Paris, France, 2001), vol. 2102 of *LNCS*, Springer, pp. 363–367.
6. Chang, K.-H., Tu, W.-T., Yeh, Y.-J., and Kuo, S.-Y. A temporal assertion extension to Verilog. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)* (2004), vol. 3299 of *LNCS*, Springer, pp. 499–504.
7. Clarke, E., Emerson, E., and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS) 8*, 2 (April 1986), 244–263.
8. Daniele, M., Giunchiglia, F., and Vardi, M. Improved automata generation for linear temporal logic. In *Conference on Computer Aided Verification (CAV)* (Trento, Italy, 1999), N. Halbwachs and D. Peled, Eds., vol. 1633 of *LNCS*, Springer, pp. 249–260.
9. Emerson, E., and Clarke, E. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming 2*, 3 (1982), 241–266.
10. Emerson, E., and Lei, C.-L. Modalities for model checking: Branching time strikes back. *Science of Computer Programming 8* (1987), 275–306.
11. Gabbay, D., Pnueli, A., Shelah, S., and Stavi, J. On the temporal analysis of fairness. In *Symposium on Principles of Programming Languages (POPL)* (New York, 1980), ACM, pp. 163–173.
12. Gastin, P., and Oddoux, D. Fast LTL to Büchi automata translation. In *Conference on Computer Aided Verification (CAV)* (Paris, France, 2001), vol. 2102 of *LNCS*, Springer, pp. 53–65.
13. Gerth, R., Peled, D., Vardi, M., and Wolper, P. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification (PSTV)* (Warsaw, June 1995), North Holland.
14. Gordon, M. PSL semantics in higher order logic. In *Workshop on Designing Correct Circuits (DCC)* (Barcelona, Spain, 2004).
15. Havlicek, J., Fisman, D., and Eisner, C. Basic results on the semantics of Accellera PSL 1.1 foundation language. Technical Report 2004.02, Accellera, 2004.
16. Kleene, S. Representation of events in nerve nets and finite automata. In *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, Princeton, NJ, 1956, pp. 3–41.
17. Landweber, L. Decision problems for ω-automata. *Mathematical Systems Theory 3*, 4 (1969), 376–384.
18. Manna, Z., and Pnueli, A. A hierarchy of temporal properties. In *Symposium on Principles of Distributed Computing* (1990), pp. 377–408.
19. Markey, N. Temporal logic with past is exponentially more succinct. *Bulletin of the European Association for Theoretical Computer Science 79* (2003), 122–128.
20. McNaughton, R., and Papert, S. *Counter-free Automata*. MIT, 1971.
21. Moorby, P. History of Verilog. *IEEE Design and Test of Computers* (September 1992), 62–63.

22. Pnueli, A. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)* (New York, 1977), vol. 18, IEEE Computer Society, pp. 46–57.
23. Reetz, R., Schneider, K., and Kropf, T. Formal specification in VHDL for formal hardware verification. In *Design, Automation and Test in Europe (DATE)* (February 1998), IEEE Computer Society.
24. Schneider, K. Improving automata generation for linear temporal logic by considering the automata hierarchy. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (Havanna, Cuba, 2001), vol. 2250 of *LNAI*, Springer, pp. 39–54.
25. Schneider, K. *Verification of Reactive Systems – Formal Methods and Algorithms.* Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.
26. Schneider, K., and Hoffmann, D. A HOL conversion for translating linear time temporal logic to omega-automata. In *Higher Order Logic Theorem Proving and its Applications (TPHOL)* (Nice, France, 1999), Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds., vol. 1690 of *LNCS*, Springer, pp. 255–272.
27. Schuele, T., and Schneider, K. Bounded model checking of infinite state systems: Exploiting the automata hierarchy. In *Formal Methods and Models for Codesign (MEMOCODE)* (San Diego, CA, June 2004), IEEE, pp. 17–26.
28. Somenzi, F., and Bloem, R. Efficient Büchi automata from LTL formulae. In *Conference on Computer Aided Verification (CAV)* (Chicago, IL, USA, 2000), E. Emerson and A. Sistla, Eds., vol. 1855 of *LNCS*, Springer, pp. 248–263.
29. Thomas, W. *Automata on Infinite Objects*, vol. B. Elsevier, 1990, ch. Automata on Infinite Objects, pp. 133–191.
30. *IEEE Standard VHDL Language Reference Manual.* New York, USA, June 1993. ANSI/IEEE Std 1076-1993.
31. Wagner, K. On ω-regular sets. *Information and Control 43* (1979), 123–177.
32. Wolper, P. Temporal logic can be more expressive. In *Symposium on Foundations of Computer Science (FOCS)* (New York, 1981), IEEE Computer Society, pp. 340–348.
33. Wolper, P., Vardi, M., and Sistla, A. Reasoning about infinite computations paths. In *Symposium on Foundations of Computer Science (FOCS)* (New York, 1983), IEEE Computer Society, pp. 185–194.

Proof Pearl: A Formal Proof of Higman's Lemma in ACL2*

Francisco J. Martín-Mateos, José L. Ruiz-Reina,
José A. Alonso, and Mariá J. Hidalgo

Computational Logic Group,
Dept. of Computer Science and Artificial Intelligence,
University of Seville, E.T.S.I. Informática,
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
http://www.cs.us.es/{~fmartin, ~jalonso, ~mjoseh, ~jruiz}

Abstract. In this paper we present a formalization and proof of Higman's Lemma in ACL2. We formalize the constructive proof described in [10] where the result is proved using a termination argument justified by the multiset extension of a well-founded relation. To our knowledge, this is the first mechanization of this proof.

1 Introduction

In [8] we presented a formal proof of Dickson's Lemma in ACL2 [6]. This result was needed to prove the termination of a Common Lisp implementation of Buchberger's algorithm for computing Gröbner basis of polynomial ideals [9]. After finishing this work our attention was addressed to similar results already present in the literature [12,15,10].The last one presents a constructive proof of Higman's Lemma similar to the one presented in [8]. Thus, the interest to automatize this proof of Higman's Lemma in ACL2 is multiple: first, the proof has a similar structure to the proof of Dickson's Lemma developed by the authors, and similar techniques are used; second, it is the first (to our knowledge) automatization of this proof, complementing thus the work presented in [10]; third, Dickson's Lemma could be proved in ACL2 as a consequence of this theorem; and finally it could give some advice about how to prove Kruskal's Theorem in ACL2, which is a fundamental theorem in the proof of termination of some well-known term orderings [1].

The ACL2 logic is a subset of first-order logic with a principle of proof by induction. The proof we present here is based on the constructive proof presented in [10], where the result is proved using a termination argument justified by the multiset extension of a well-founded relation. In the mechanization of this proof, we use a tool for defining multiset well-founded relations in ACL2 in an automated way, a tool that we used previously in other formalizations [13] and that can now be reused.

* This work has been supported by project TIN2004-03884 (Ministerio de Educación y Ciencia, Spain) and FEDER founds.

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 358–372, 2005.

Higman's Lemma is a property about embeddings of strings. Previously to present the result we introduce some notation. Let Σ be a set, and let Σ^* denote the set of finite strings over Σ.

Definition 1. *Let $\preceq$ be a binary relation on Σ. The* induced embedding relation *$\preceq^*$ on Σ^* is defined as follows: $s_1 s_2 \cdots s_m \preceq^* t_1 t_2 \cdots t_n$ if there exists indices $j_1 < j_2 < \ldots < j_m \leq n$ such that $s_i \preceq t_{j_i}, \forall i$.*

If $s \preceq t$ ($u \preceq^* w$) we usually say that s (u) is less than t (w) or t (w) is bigger than s (u). The relation with respect to which an element is less or bigger than other is usually obvious in the context.

Definition 2. *We say that a relation on Σ is a* quasi-order *if it is reflexive and transitive. Given a quasi-order $\preceq$ defined on Σ, we say that $\preceq$ is a* well-quasi-order *if for every infinite sequence*[1] *$\{s_k : k \in \mathbb{N}\}$ of elements of Σ there exist indices $i < j$ such that $s_i \preceq s_j$.*

Higman's Lemma establishes a sufficient condition for well-quasi-orders on strings.

Theorem 1. (Higman's Lemma). *If $\preceq$ is a well-quasi-order on Σ then $\preceq^*$ is also a well-quasi-order on Σ^*.*

Given the well-quasi-order $\preceq$ on Σ, it is not difficult to prove that $\preceq^*$ is a quasi-order on Σ^*. Thus, we will center our attention on the well-quasi-order property: for every infinite sequence of strings $\{w_k : k \in \mathbb{N}\}$ there exists indices $i < j$ such that $w_i \preceq^* w_j$.

As we said above, the proof presented here is based on [10], and it essentially builds a well-founded measure that can be associated to the initial segments of a sequence of strings and that decreases whenever a string in the sequence is not bigger than any of the previous strings.

2 Formalizing the Proof in ACL2

The ACL2 logic is a first-order logic with equality, describing an applicative subset of Common Lisp. The syntax of terms is that of Common Lisp and the logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference of the logic include those for propositional calculus, equality and instantiation. One important rule of inference is the *principle of induction*, that permits proofs by well-founded induction on the ordinal ε_0. The theory has a constructive definition of the ordinals up to ε_0, in terms of lists and natural numbers, given by the predicate `o-p` ($o \in \varepsilon_0 \equiv$ `o-p`(o)) and the order ($o_1 <_{\varepsilon_0} o_2 \equiv$ `o<`(o_1,o_2).) Although this is the only built-in well-founded relation, the user may define new well-founded relations from that.

[1] An infinite sequence of elements of Σ is a function $s : \mathbb{N} \to \Sigma$. As usual, we write s_k instead of $s(k)$ and by abuse of notation, we often identify the sequence with its range $\{s_k : k \in \mathbb{N}\}$.

By the *principle of definition*, new function definitions are admitted as axioms only if there exists a measure in which the arguments of each recursive call decrease with respect to a well-founded relation, ensuring in this way that no inconsistencies are introduced by new definitions. Usually, the system can prove automatically this property using a predefined ordinal measure on Lisp objects and the relation `o<`. Nevertheless, if the termination proof is not trivial, the user has to explicitly provide a measure on the arguments and a well-founded relation ensuring termination.

The `encapsulate` mechanism [7] allows the user to introduce new function symbols by axioms constraining them to have certain properties (to ensure consistency, a witness local function having the same properties has to be exhibited). Inside an `encapsulate`, the properties stated need to be proved for the local witnesses, and outside, they work as assumed axioms. This mechanism behaves like a universal quantifier over a set of functions abstractly defined with it.

A derived rule of inference, called *functional instantiation*, provides a higher-order-like mechanism by allowing to instantiate the function symbols of a previously proved theorem, replacing them with other function symbols or lambda expressions, provided it can be proved that the new functions satisfy the constraints of the old ones.

The ACL2 theorem prover mechanizes the ACL2 logic, being particularly well suited for obtaining automatized proofs based on simplification and induction. For a detailed description of ACL2, we refer the reader to the ACL2 book [5].

For the sake of readability, the ACL2 expressions in this paper are presented using a notation closer to the usual mathematical notation than its original Common Lisp syntax; when it is necessary we show the correspondence between the ACL2 and the mathematical notation. Some of the functions are also used in infix notation.

2.1 Formulation of Higman's Lemma

First we formalize in the ACL2 logic the context in which Higman's Lemma will be established. We consider a unary predicate `sigma-p` to check the membership to Σ ($s \in \Sigma \equiv$ `sigma-p`(s)) and a binary predicate `sigma-<=` representing the well-quasi-order $\preceq$ on the set Σ ($s \preceq t \equiv$ `sigma-<=`(s,t)). These functions are abstractly defined by means of the `encapsulate` mechanism. In this case the assumed properties about `sigma-p` and `sigma-<=` are the following[2]:

ASSUMPTION: `sigma-<=-reflexive`
$s \in \Sigma \rightarrow s \preceq s$

ASSUMPTION: `sigma-<=-transitive`
$s_1, s_2, s_3 \in \Sigma \wedge s_1 \preceq s_2 \wedge s_2 \preceq s_3 \rightarrow s_1 \preceq s_3$

In the following we use $\overline{\Sigma}$ to denote the set of finite sequences of elements of Σ and we use the overline notation to identify the elements of $\overline{\Sigma}$. We characterize

[2] The local witnesses are irrelevant to our description of the proof.

the well-quasi-order property of $\preceq$ in the following way: there exists an ordinal measure on $\overline{\Sigma}$ such that the measure of any element of $\overline{\Sigma}$ is bigger than the measure of any extension of this sequence with an element s such that there is no elements in the sequence less than s. As it is pointed out in [10]: "*Classically, this is easily gotten from the well-quasi-order-ness of* $\preceq$*, but constructively we must have this as an assumption (...) After a moment's reflection, it should be obvious to the reader that* this *is the constructive equivalent to the classical notion of well-quasi-order.*"

The embedding is introduced in the previous `encapsulate` by means of the function `sigma-seq-measure`. To state the properties assumed about this function we also need two concepts whose definitions are based on `sigma-p` and `sigma-<=` functions: the membership to $\overline{\Sigma}$ ($\overline{s} \in \overline{\Sigma} \equiv$ `sigma-seq-p`$(\overline{s})$) and the presence of an element in a finite sequence $\overline{s}$ less than an element t (`exists-sigma-<=`$(\overline{s},t)$). The assumed properties are the following:

ASSUMPTION: `sigma-seq-measure-o-p`
$\overline{s} \in \overline{\Sigma} \rightarrow$ `sigma-seq-measure`$(\overline{s}) \in Ord$

ASSUMPTION: `sigma-seq-measure-neq-0`
$\overline{s} \in \overline{\Sigma} \rightarrow$ `sigma-seq-measure`$(\overline{s}) \neq 0$

ASSUMPTION: `sigma-<=-well-quasi-order-characterization`
$\overline{s} \in \overline{\Sigma} \wedge\ t \in \Sigma \wedge \neg$`exists-sigma-<=`$(\overline{s},t)$
$\rightarrow$ `sigma-seq-measure`$(\mathtt{cons}(t,\overline{s})) <_{\varepsilon_0}$ `sigma-seq-measure`$(\overline{s})$

The first property ensures that the function `sigma-seq-measure` returns an ACL2 ordinal when its argument is a string, and the second property ensures that this ordinal is not 0[3]. The last property is the constructive characterization of the well-quasi-order-ness of $\preceq$. It must be noticed that the elements of $\overline{\Sigma}$ are represented in ACL2 by means of lists of elements of Σ in reverse order; that is, the ACL2 representation of the finite sequence $\{s_1, \ldots, s_n\}$ is the list $(s_n \ \ldots \ s_1)$. This is because an element t is more easily added in front of a sequence $\overline{s}$ using `cons`.

The elements of Σ^* are also represented in ACL2 by means of lists, but in this case the order of the elements is preserved. So, the representation of the string $s_1 s_2 \ldots s_m$ is the list $(s_1\ s_2\ \ldots\ s_m)$. The membership to Σ^* is checked by the function `sigma-*-p` ($w \in \Sigma^* \equiv$ `sigma-*-p`(w)). We use a different function name, but its definition is equal to the definition of `sigma-seq-p`. The relation $\preceq^*$ in Σ^* is formalized with the following function ($w_1 \preceq^* w_2 \equiv$ `sigma-*-<=`(w_1,w_2)):

DEFINITION:
`sigma-*-<=`$(w_1,w_2) \Leftrightarrow$
if `endp`(w_1) **then t**

[3] This is a technical detail that could have been avoided adding 1 to the finite ordinals returned by `sigma-seq-measure`.

elseif endp(w_2) **then nil**
elseif car(w_1) $\in \Sigma \wedge$ car(w_1) $\in \Sigma$
then if car(w_1) $\preceq$ car(w_2)
then sigma-*-<=(cdr(w_1),cdr(w_2))
else sigma-*-<=(w_1,cdr(w_2))
else nil

It should be noted that this definition is algorithmic and different from the non-deterministic declarative Definition 1; but it is not difficult to prove that both definitions are equivalent. This function checks the property $w_1 \preceq^* w_2$ looking for the first elements, from the left side of w_2, bigger than the elements of w_1.

To formalize Higman's Lemma in the ACL2 logic we consider a unary function f, representing an infinite sequence of strings. We use again the encapsulate mechanism to abstractly define this function. In this case, the assumed property about f is the following:

ASSUMPTION: f-returns-strings
$i \in \mathbb{N} \rightarrow$ f(i) $\in \Sigma^*$

Here, the encapsulate mechanism behaves like a universal quantifier over the function abstractly defined with it. So, any theorem proved about this function is true for any function with the above property, by means of functional instantiation (see [5] for details). This is the case for the ACL2 formalization of Higman's Lemma: as the infinite sequence of strings is abstractly defined via encapsulate, the properties that we will prove about it are valid for any infinite sequence of strings.

Let us now define the functions needed to state Higman's Lemma. The function get-sigma-*-<=-f has two arguments, a natural number j and a string w, and it returns the biggest index i such that $i < j$ and f(i) $\preceq^* w$ whenever such index exists (nil otherwise):

DEFINITION:
get-sigma-*-<=-f(j,w) =
if $j \in \mathbb{N}$ **then if** $j = 0$ **then nil**
elseif f($j-1$) $\preceq^* w$ **then** $j-1$
else get-sigma-*-<=-f($j-1$,w)
else nil

Finally, the following function higman-indices receives as input an index k and uses get-sigma-*-<=-f to recursively search a pair of indices $i < j$ such that $j \geq k$ and f(i) $\preceq^*$ f(j):

DEFINITION:
higman-indices(k) =
if $k \in \mathbb{N}$ **then let** i **be** get-sigma-*-<=-f(k,f(k))
in if $i \neq$ **nil then** $\langle i, k \rangle$
else higman-indices($k+1$)
else nil

Let us assume for the moment that we have proved that the function `higman-indices` terminates and that this definition has been admitted by the system. Then the following property is easily proved as a direct consequence of the definitions of the functions involved:

THEOREM: `higman-lemma`
$[k \in \mathbb{N} \wedge \texttt{higman-indices}(k) = \langle i, j \rangle] \rightarrow [i < j \wedge \texttt{f}(i) \preceq^* \texttt{f}(j)]$

This theorem ensures that for any infinite sequence of strings $\{\texttt{f}(k) : k \in \mathbb{N}\}$, there exists $i < j$ such that $\texttt{f}(i) \preceq^* \texttt{f}(j)$ (and the function `higman-indices` explicitly provides those indices). Thus, it is a formal statement of Higman's Lemma in ACL2.

The hard part is the termination proof of the function `higman-indices`. For that purpose, we have to explicitly provide to the system a measure on the input argument and prove that the measure decreases with respect to a given well-founded relation in every recursive call. We present the details in the next subsections.

2.2 A Well-Founded Measure

Before giving a formal definition of the termination measure, we give some intuition by means of an example. Let us consider the set of natural numbers, $\mathbb{N}$, and the reflexive transitive closure of the following relation: $n \preceq n+2$ for all n. In this relation the even numbers are ordered as usual, as well as the odd numbers, but there is no relation between even and odd numbers. It is easy to prove that this relation is a well-quasi-order.

An embedding from $\overline{\mathbb{N}}$ in ordinals characterizing the well-quasi-order-ness of $\preceq$ could be the following: given a finite sequence of natural numbers $\overline{s}$, the ordinal associated is $\omega \cdot 2$ if $\overline{s}$ is empty; $\omega + n$ if n is the last even (odd) number in $\overline{s}$ and there is no odd (even) numbers in $\overline{s}$; and $n + m$ if n and m are respectively the last even number and the last odd number in $\overline{s}$.

Let $\{f_k : k \in \mathbb{N}\}$ be an infinite sequence of strings over $\mathbb{N}$ with $f_0 = 3 \cdot 2$. Note that any string bigger than f_0 is of the form $x_1 ... x_i x y_1 ... y_j y z_1 ... z_k$ with $3 \preceq x$ and $2 \preceq y$. Thus, the strings w such that $f_0 \not\preceq^* w$ could be described as follows:

- Any string $x_1 ... x_n$ with $3 \not\preceq x_i, \forall i$. We represent this set of strings by $\Pi_1 = \langle -, 3 \rangle$.
- Any string $x_1 ... x_n x y_1 ... y_m$ with $3 \not\preceq x_i, \forall i$, $3 \preceq x$ and $2 \not\preceq y_j, \forall j$. There are three components in these strings: $x_1 ... x_n$ with $3 \not\preceq x_i$, that we represent by $\pi_1 = \langle -, 3 \rangle$; x with $3 \preceq x$, that we represent by $\pi_2 = \langle 3 \rangle$; and $y_1 ... y_m$ with $2 \not\preceq y_j$, that we represent by $\pi_3 = \langle -, 2 \rangle$. So the representation of this set of strings is $\Pi_2 = \pi_1 \pi_2 \pi_3 = \langle -, 3 \rangle \langle 3 \rangle \langle -, 2 \rangle$.

We will refer to these representations of set of strings as *patterns*. As it can be seen in the previous example, a pattern could have components that we will call *simple patterns*. Π_1 is a pattern with only one simple pattern $\langle -, 3 \rangle$; and Π_2 is a pattern with three simple patterns π_1, π_2 and π_3. We will use the capital

Greek letter Π (possibly with subscripts) to represent patterns, and the small Greek letter π (possibly with subscripts) to represent simple patterns. The set of strings represented by a pattern Π or a simple pattern π will be denoted $\mathcal{S}(\Pi)$ and $\mathcal{S}(\pi)$ respectively.

Let us suppose that $f_1 = 1 \cdot 2$. Note that $f_1 \in \mathcal{S}(\Pi_1)$ since every element of f_1 is not bigger than 3. Let us see now how we can obtain from Π_1 a set of patterns describing the set of strings w such that $f_0 \not\preceq^* w$ and $f_1 \not\preceq^* w$ (we will call this operation the *reduction* of Π_1 with respect to f_1). The pattern Π_1 should be replaced in such a way that any string bigger than f_1 is removed from the set represented by the pattern. Since the strings bigger than f_1 are of the form $x_1...x_i x y_1...y_j y z_1...z_k$ with $1 \preceq x$ and $2 \preceq y$, then the set of strings $w \in \mathcal{S}(\Pi_1)$ such that $f_1 \not\preceq^* w$ could be described as follows:

- Any string $x_1...x_n$ with $3 \not\preceq x_i, \forall i$ (the string is in $\mathcal{S}(\Pi_1)$) and $1 \not\preceq x_i, \forall i$. We represent this set of strings by $\langle -, 3, 1\rangle$.
- Any string $x_1...x_n x y_1...y_m$ with $3 \not\preceq x_i, x, y_j, \forall i, j$ (the string is in $\mathcal{S}(\Pi_1)$), $1 \not\preceq x_i, \forall i$, $1 \preceq x$ and $2 \not\preceq y_j, \forall j$. There are three components in these strings: $x_1...x_n$ with $3, 1 \not\preceq x_i, \forall i$, represented by $\langle -, 3, 1\rangle$; x with $3 \not\preceq x$ and $1 \preceq x$, represented by $\langle 1, 3\rangle$; and $y_1...y_m$ with $3, 2 \not\preceq y_j, \forall j$, represented by $\langle -, 3, 2\rangle$. So the representation of this set of strings is $\langle -, 3, 1\rangle\langle 1, 3\rangle\langle -, 3, 2\rangle$.

Now, let us see how to deal with a more complicated pattern: let suppose that $f_2 = 1 \cdot 5 \cdot 3$; this string is in $\mathcal{S}(\Pi_2)$ since $1 \in \mathcal{S}(\pi_1)$, $5 \in \mathcal{S}(\pi_2)$ and $3 \in \mathcal{S}(\pi_3)$. The pattern Π_2 should be reduced in such a way that any string bigger than f_2 is removed. In this case if $w = w_1 w_2 w_3 \in \mathcal{S}(\Pi_2)$ with $w_i \in \mathcal{S}(\pi_i)$ and $1 \preceq^* w_1$, $5 \preceq^* w_2$ and $3 \preceq^* w_3$, then $f_2 \preceq^* w$. Thus, if we ensure that some component w_i is not bigger than the correspondent component in f_2, then $f_2 \not\preceq^* w$. Taking this into account, the strings $w \in \mathcal{S}(\Pi_2)$ such that $f_2 \not\preceq^* w$ are described as follows:

- Any string $w_1 w_2 w_3$ with $w_i \in \mathcal{S}(\pi_i), \forall i$ and $1 \not\preceq^* w_1$. The set of strings in $\mathcal{S}(\pi_1)$ whose elements are not bigger than 1 is represented by the pattern $\langle -, 3, 1\rangle$. So the representation of this set of strings is $\langle -, 3, 1\rangle\langle 3\rangle\langle -, 2\rangle$.
- Any string $w_1 w_2 w_3$ with $w_i \in \mathcal{S}(\pi_i), \forall i$ and $5 \not\preceq^* w_2$. The set of strings in $\mathcal{S}(\pi_2)$ whose elements are not bigger than 5 is represented by the pattern $\langle 3, 5\rangle$. So the representation of this set of strings is $\langle -, 3\rangle\langle 3, 5\rangle\langle -, 2\rangle$.
- Any string $w_1 w_2 w_3$ with $w_i \in \mathcal{S}(\pi_i), \forall i$ and $3 \not\preceq^* w_3$. The set of strings in $\mathcal{S}(\pi_3)$ whose elements are not bigger than 3 is represented by the pattern $\langle -, 2, 3\rangle$. So the representation of this set of strings is $\langle -, 3\rangle\langle 3\rangle\langle -, 2, 3\rangle$.

We now explain how we formalize these constructions in ACL2. There are two types of simple patterns: the first one is $\langle -, s_1, \ldots, s_n\rangle$(`(nil` s_n `...` s_1`)` in ACL2), representing any string $t_1 \ldots t_m$ such that $s_i \not\preceq t_j, \forall i, j$; the second one is $\langle s, s_1, \ldots, s_n\rangle$ (`((`s`)` s_n `...` s_1`)` in ACL2), representing any string t such that $s \preceq t$ and $s_i \not\preceq t, \forall i$ (π is a simple pattern $\equiv$ `simple-pattern-p`(π)). In both cases we say that $s_1, \ldots, s_n$ is the sequence with respect to which the simple pattern is defined. Note that this sequence is represented in ACL2 in reverse order.

The following function checks the membership of a 1-length string s to the set of strings described by a simple pattern π:

DEFINITION:
`member-simple-pattern`$(s,\pi) =$
  if `consp`(`car`(π)) **then** `caar`(π) $\preceq s \wedge \neg$`exists-sigma-<=`(`cdr`(π),s)
  else $\neg$`exists-sigma-<=`(`cdr`(π),s)

Simple patterns represent the components in which a new string could be split ensuring that it is not bigger than any previous string in the sequence. Thus, a pattern is a string of simple patterns (Π is a pattern $\equiv$ `pattern-p`(Π)). Given $\Pi = \pi_1 \ldots \pi_n$, a string w is in the set of strings described by Π ($w \in \mathcal{S}(\Pi)$) if there exist strings $w_1, \ldots, w_n$, such that $w = w_1 \ldots w_n$ and $w_i \in \mathcal{S}(\pi_i), \forall i$. The function `member-pattern`$(w,\Pi)$ returns a pair `(res val)` where `res` indicates if $w \in \mathcal{S}(\Pi)$ and, if it is the case, `val` is the list of components `(`w_1 `...` w_n`)` justifying this.

DEFINITION:
`member-pattern`$(w,\Pi) =$
  if `endp`(w) **then if** `endp`(Π) **then** $\langle$**t**, **nil**$\rangle$
                  elseif `consp`(`caar`(Π)) **then** $\langle$**nil**, **nil**$\rangle$
                  else let $\langle$`res`, `val`$\rangle$ **be** `member-pattern`(w,`cdr`(Π))
                    in if `res` **then** $\langle$**t**, `cons`(**nil**, `val`)$\rangle$
                      else $\langle$**nil**, **nil**$\rangle$
  elseif `endp`(Π) **then** $\langle$**nil**, **nil**$\rangle$
  elseif `consp`(`caar`(Π))
      let `res1` **be** `member-simple-pattern`(`car`(w),`car`(Π))
        $\langle$`res2`, `val2`$\rangle$ **be** `member-pattern`(`cdr`(w),`cdr`(Π))
      in if `res1` $\wedge$ `res2` **then** $\langle$**t**, `cons`(`list`(`car`(w)), `val2`)$\rangle$
        else $\langle$**nil**, **nil**$\rangle$
  else let $\langle$`res1`, `val1`$\rangle$ **be** `member-pattern`(w,`cdr`(Π))
      in if `res1` **then** $\langle$**t**, `cons`(**nil**, `val1`)$\rangle$
        else let `res2` **be** `member-simple-pattern`(`car`(w),`car`(Π))
          $\langle$`res3`, `val3`$\rangle$ **be** `member-pattern`(`cdr`(w),Π)
        in if `res2` $\wedge$ `res3`
            then $\langle$**t**, `cons`(`cons`(`car`(w), `car`(`val3`)), `cdr`(`val3`))$\rangle$
          else $\langle$**nil**, **nil**$\rangle$

As we said above, given a string w and a pattern Π such that $w \in \mathcal{S}(\Pi)$, we define the reduction of Π with respect to w as a set of patterns representing the strings in $\mathcal{S}(\Pi)$ not bigger than w. Let us describe the reduction process beginning with the reductions of a simple pattern π:

- If $\pi = \langle s, s_1, \ldots, s_n \rangle$ then $w \in \Sigma$, $s \preceq w$ and $s_i \not\preceq w, \forall i$. In this case the pattern $\langle s, s_1, \ldots, s_n, w \rangle$ represents the set of strings t such that $s \preceq t$, $s_i \not\preceq t, \forall i$ and $w \not\preceq t$, that is, the strings in $\mathcal{S}(\pi)$ that are not bigger than w.
- If $\pi = \langle -, s_1, \ldots, s_n \rangle$ then $w = t_1 \ldots t_m$ such that $s_i \not\preceq t_j, \forall i, j$. In this case we obtain the following patterns after reducing π:

- $\langle -, s_1, \ldots, s_n, t_1 \rangle$: This pattern represents any string w whose elements are not bigger than $s_i, \forall i$ and t_1.
- $\langle -, s_1, ..., s_n, t_1 \rangle \langle t_1, s_1, ..., s_n \rangle \langle -, s_1, ..., s_n, t_2 \rangle$: This pattern represents any string $w_1 t w_2$ such that, the elements of w_1 are not bigger than $s_i, \forall i$ and t_1; t is an element of Σ bigger than t_1 and not bigger than $s_i, \forall i$; and the elements of w_2 are not bigger than $s_i, \forall i$ and t_2.
- $\langle -, s_1, ..., s_n, t_1 \rangle \langle t_1, s_1, ..., s_n \rangle \langle -, s_1, ..., s_n, t_2 \rangle \langle t_2, s_1, ..., s_n \rangle \langle -, s_1, ..., s_n, t_3 \rangle$
- And so on.

The patterns obtained in the second case are all disjoint because the first one represents strings whose elements are not bigger than t_1, the second one represents strings with an element bigger than t_1 followed by elements not bigger than t_2, the third one represents strings with an element bigger than t_1 followed by an element bigger than t_2 and this followed by elements not bigger than t_3, and so on. This disjointness property is preserved by the reduction process of general patterns.

The function `reduce-simple-pattern`(w,π) computes the reductions of the simple pattern π with respect to the string w (assuming that $w \in \mathcal{S}(\pi)$). Let us recall that the sequence with respect to which a simple pattern is defined is represented in ACL2 in reverse order, allowing easy additions of new elements to it:

DEFINITION:
`reduce-simple-pattern`(w,π) =
  if `endp`(w) **then nil**
  elseif `consp`(`car`(π))
    then `list`(`list`(`cons`(`car`(π),`cons`(`car`(w),`cdr`(π))))))
  else `cons`(`list`(`cons`(**nil**,`cons`(`car`(w),`cdr`(π)))),
    `cons2-list-cdr`(`cons`(**nil**,`cons`(`car`(w),`cdr`(π))),
      `cons`(`list`(`car`(w)),`cdr`(π)),
      `reduce-simple-pattern`(`cdr`(w),π)))

where the function `cons2-list-cdr` behaves schematically in the following way:

`(cons2-list-cdr 'x 'y '(`l_1 `...` l_n`)) = '((`x y `.` l_1`) ... (`x y `.` l_n`))`

As we have discussed in the example, the reduction of a pattern depends on its components. Given a pattern $\Pi = \pi_1 \ldots \pi_n$ and a string $w \in \mathcal{S}(\Pi)$, there exist strings $w_1 \ldots w_n$ such that $w = w_1 \ldots w_n$ and $w_i \in \mathcal{S}(\pi_i), \forall i$. The reduction of Π with respect to w is the set of patterns $\pi_1 \ldots \pi_{i-1} \pi'_1 \ldots \pi'_m \pi_{i+1} \ldots \pi_n$ for every index i and every pattern $\pi'_1 \ldots \pi'_m$ obtained by reducing the simple pattern π_i with respect to w_i. The function `reduce-simple-pattern-list` computes the reduction of a list of simple patterns (the components of Π) with respect to a list of strings (the components of w).

DEFINITION:
`reduce-simple-pattern-list`(w-lst,Π) =
  if `endp`(Π) **then nil**

else append-list-car(reduce-simple-pattern(car(w-lst),car(Π)),
cdr(Π)) @
cons-list-cdr(car(Π),
reduce-simple-pattern-list(cdr(w-lst),cdr(Π)))

where the symbol @ is the "append" operation between lists, and the functions append-list-car and cons-list-cdr behave schematically as follows:

(append-list-car '(l_1 ... l_n) 'l = '(l_1 @ l ... l_n @ l)
(cons-list-cdr 'x '(l_1 ... l_n)) = '((x . l_1) ... (x . l_n))

The function reduce-pattern(w,Π) computes the reduction of the pattern Π with respect to the string w, whenever $w \in \mathcal{S}(\Pi)$. If this is not the case, the function returns the initial pattern Π:

Definition:
reduce-pattern(w,Π) =
let ⟨res, val⟩ **be** member-pattern(w,Π)
in if res **then** reduce-simple-pattern-list(val,Π)
else Π

Now we deal with set of patterns. In the following let be denote $\mathcal{P}$ a set of patterns and $\mathcal{S}(\mathcal{P})$ the set of strings represented by the patterns in $\mathcal{P}$ ($\mathcal{P}$ is a set of patterns ≡ pattern-list-p($\mathcal{P}$)). The function reduce-pattern-list describes how the set of patterns $\mathcal{P}$ is reduced with respect to a string w:[4]

Definition:
reduce-pattern-list(w,$\mathcal{P}$) =
if endp($\mathcal{P}$) **then** $\mathcal{P}$
else let ⟨res, val⟩ **be** member-pattern(w,car($\mathcal{P}$))
in if res **then** reduce-pattern(w,car($\mathcal{P}$)) @ cdr($\mathcal{P}$)
else cons(car($\mathcal{P}$),reduce-pattern-list(w,cdr($\mathcal{P}$)))

The function reduce-pattern-sequence-list iterates the reduction process over a finite sequence of strings. It must be noticed that the sequence of strings is provided in reverse order:

Definition:
reduce-pattern-sequence-list($\overline{w}$,$\mathcal{P}$) =
if endp($\overline{w}$) **then** $\mathcal{P}$
else reduce-pattern-list(
car($\overline{w}$),reduce-pattern-sequence-list(cdr($\overline{w}$),$\mathcal{P}$))

[4] Since the reduction process produces patterns representing disjoint sets of strings, it is enough to make the reduction with respect to the first pattern matched by member-pattern. It must be noticed that it would be sufficient to make one reduction even in the case in which $\mathcal{P}$ contains patterns representing non-disjoint sets of strings.

Recall that our goal is to assign a well-founded measure to every initial subsequence of `f`, in such a way that every time a new element appears in the sequence such that it is not bigger than any of the previous elements, then the corresponding measure of the extended subsequence is strictly smaller. Intuitively, we will measure the "size" of the set of elements that can be the next in the sequence without affecting the well-quasi-order-ness property. Since we have seen that this set of strings can be represented by a set of patterns, we will measure sets of patterns.

First, we assign an ordinal measure to simple patterns, based on the ordinal measure provided by `sigma-seq-measure`: if o is the measure of the sequence $\overline{s}$, then the measure of the simple pattern $\langle -, \overline{s} \rangle$ is $\omega^o + 1$, and the measure of the simple pattern $\langle t, \overline{s} \rangle$ is ω^o. The measure of a pattern is the multiset of the measures of the simple patterns in it. Finally, the measure of a set of patterns is the multiset of measures of the patterns in it. The ACL2 functions `simple-pattern-measure`(π), `pattern-measure`(Π) and `pattern-list-measure`($\mathcal{P}$) compute respectively the measure of the simple pattern π, of the pattern Π and of the set of patterns $\mathcal{P}$. We use this last function to associate a measure to every index k:[5]

DEFINITION:

```
higman-indices-measure(k) =
  pattern-list-measure(
    reduce-pattern-sequence-list(initial-segment-f(k − 1),
                                 list(initial-pattern())))
```

where the function `initial-pattern`() builds the initial pattern $\langle - \rangle$ and the function `initial-segment-f`(k) builds the list of strings `(`f_k ... f_1 f_0`)`. Note that this measure is a finite multiset of finite multisets of ordinals.

The following table summarizes the measures in the given example:

Initial subsequence	Set of patterns	Measure
$\{\}$	$\{\langle - \rangle\}$	$\{\!\{\{\!\{\omega^{\omega \cdot 2} + 1\}\!\}\}\!\}$
$\{3 \cdot 2\}$	$\{\langle -, 3 \rangle,$ $\langle -, 3 \rangle\langle 3 \rangle\langle -, 2 \rangle\}$	$\{\!\{\{\!\{\omega^{\omega+3} + 1\}\!\},$ $\{\!\{\omega^{\omega+3} + 1, \omega^{\omega \cdot 2}, \omega^{\omega+2} + 1\}\!\}\}\!\}$
$\{3 \cdot 2,$ $1 \cdot 2\}$	$\{\langle -, 3, 1 \rangle,$ $\langle -, 3, 1 \rangle\langle 1, 3 \rangle\langle -, 3, 2 \rangle,$ $\langle -, 3 \rangle\langle 3 \rangle\langle -, 2 \rangle\}$	$\{\!\{\{\!\{\omega^{\omega+1} + 1\}\!\},$ $\{\!\{\omega^{\omega+1} + 1, \omega^{\omega+3}, \omega^{3+2} + 1\}\!\},$ $\{\!\{\omega^{\omega+3} + 1, \omega^{\omega \cdot 2}, \omega^{\omega+2} + 1\}\!\}\}\!\}$
$\{3 \cdot 2,$ $1 \cdot 2,$ $1 \cdot 5 \cdot 3\}$	$\{\langle -, 3, 1 \rangle,$ $\langle -, 3, 1 \rangle\langle 1, 3 \rangle\langle -, 3, 2 \rangle,$ $\langle -, 3, 1 \rangle\langle 3 \rangle\langle -, 2 \rangle,$ $\langle -, 3 \rangle\langle 3, 5 \rangle\langle -, 2 \rangle,$ $\langle -, 3 \rangle\langle 3 \rangle\langle -, 2, 3 \rangle\}$	$\{\!\{\{\!\{\omega^{\omega+1} + 1\}\!\},$ $\{\!\{\omega^{\omega+1} + 1, \omega^{\omega+3}, \omega^{3+2} + 1\}\!\},$ $\{\!\{\omega^{\omega+1} + 1, \omega^{\omega \cdot 2}, \omega^{\omega+2} + 1\}\!\},$ $\{\!\{\omega^{\omega+3} + 1, \omega^{\omega+5}, \omega^{\omega+2} + 1\}\!\},$ $\{\!\{\omega^{\omega+3} + 1, \omega^{\omega \cdot 2}, \omega^{3+2} + 1\}\!\}\}\!\}$

[5] Note that since `f` is fixed, then k is sufficient to represent the initial subsequence $\{f_0 \ldots f_{k-1}\}$.

2.3 Termination Proof of `higman-indices`

The last step in this formal proof is to define a well-founded relation and prove that the given measure decreases with respect to it in every recursive call of the function `higman-indices`. We will define it as the relation on finite multisets induced by a well-founded relation. Intuitively, this relation is defined in such a way that a smaller multiset can be obtained by removing a non-empty subset of elements, and adding elements which are smaller than some of the removed elements. In [4], Dershowitz and Manna show that if the base relation is well-founded, then the relation induced on finite multisets is also well-founded.

As we said above, the only predefined well-founded relation in ACL2 is `o<`, implementing the usual order between ordinals less than ε_0. The function `o-p` recognizes those ACL2 objects representing such ordinals. If we want to define a new well-founded relation in ACL2, we have to explicitly provide a monotone ordinal function, and prove the corresponding order-preserving theorem (see [5] for details). Fortunately, we do not have to do this: we use the `defmul` tool. This tool, previously implemented and used by the authors in [13], automatically generates the definitions and proves the theorems needed to introduce in ACL2 the multiset relation induced by a given well-founded relation.

In our case, we need two `defmul` calls. The first one automatically generates the definition of the function `mul-o<`, implementing the multiset relation on finite multisets of ordinals (the measure of a pattern) induced by the relation `o<`; and the second one automatically generates the definition of `mul-mul-o<`, implementing the multiset relation of finite multisets of finite multisets of ordinals (the measure of a pattern list) induced by the relation `mul-o<`. These calls also automatically prove the theorems needed to introduce these relations as well-founded relations in ACL2. See details about the `defmul` syntax in [13]. For simplicity, in the following we denote `mul-o<` as $<_{\varepsilon_0,\mathcal{M}}$ and `mul-mul-o<` as $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$.

We finally prove that the measure decreases with respect to $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$ in the recursive call of the function `higman-indices`, hence justifying its termination. We now explain the main lemmas needed to prove this result. We start with the ones related to simple patterns. We have two cases:

- If the simple pattern is $\pi = \langle s, s_1, \ldots, s_n \rangle$ and a string $w \in \mathcal{S}(\pi)$, then $w \in \Sigma$, $s \preceq w$ and $s_i \not\preceq w, \forall i$. In this case the reduction of π with respect to w is the simple pattern $\pi' = \langle s, s_1, \ldots, s_n, w \rangle$. Then $\neg$`exists-sigma-<=`$(\overline{s},w)$ where $\overline{s} = \langle s_1, \ldots, s_n \rangle$. Thus the well-quasi-order characterization of $\preceq$ ensures that `sigma-seq-measure`(`cons`$(w,\overline{s})$) is less than `sigma-seq-measure`$(\overline{s})$. Therefore, the measure of π' is less than the measure of π.
- If the simple pattern is $\pi = \langle -, s_1, \ldots, s_n \rangle$ and a string $w \in \mathcal{S}(\pi)$, then $w = t_1 \ldots t_m$ such that $s_i \not\preceq t_j, \forall i, j$. In this case the reduction of π with respect to w is a set of patterns whose components are of one of two kinds:
  - Simple patterns as $\pi' = \langle -, s_1, \ldots, s_n, t_j \rangle$. Then, if $\overline{s} = \langle s_1, \ldots, s_n \rangle$, we have $\neg$`exists-sigma-<=`$(\overline{s},t_j)$. Thus the well-quasi-order characterization of $\preceq$ ensures that `sigma-seq-measure`(`cons`$(t_j,\overline{s})$) is less than

`sigma-seq-measure`($\overline{s}$). Therefore, the measure of π' is less than the measure of π.

- Simple patterns as $\pi' = \langle t_j, s_1, \ldots, s_n \rangle$. Then, if $\overline{s} = \langle s_1, \ldots, s_n \rangle$ and $o =$`sigma-seq-measure`($\overline{s}$), the measure of π' is ω^o and the measure of π is $\omega^o + 1$. Therefore, the measure of π' is less than the measure of π.

The following ACL2 lemma summarizes these considerations:

LEMMA: `reduce-simple-pattern-property`
(`simple-pattern-p`(π) $\wedge$ $w \in \Sigma^*$ $\wedge$ $w \in \mathcal{S}(\pi)$
$\wedge$ $\Pi \in$ `reduce-simple-pattern`(w,π) $\wedge$ $\pi' \in \Pi$)
$\rightarrow$ `measure-simple-pattern`(π') $<_{\varepsilon_0}$ `measure-simple-pattern`(π)

where $\pi' \in \Pi$ indicates that the simple pattern π' is a component of the pattern Π.

Given a pattern $\Pi = \pi_1 \ldots \pi_n$ and $w \in \mathcal{S}(\Pi)$, to compute the patterns in the reduction of Π with respect to w, we consider $w = w_1 \ldots w_n$ such that $w_i \in \mathcal{S}(\pi_i), \forall i$, and $\pi'_1 \ldots \pi'_m$ obtained by reducing the simple pattern π_i with respect to w_i. Then, the pattern $\Pi' = \pi_1 \ldots \pi_{i-1} \pi'_1 \ldots \pi'_m \pi_{i+1} \ldots \pi_n$ is in the reduction of Π with respect to w. The previous lemma ensures that the measure of π'_j is less than the measure of $\pi_i, \forall i, j$, therefore, the measure of Π' is obtained from the measure of Π, replacing the measure of π_i with the smaller measures of $\pi'_1, \ldots, \pi'_m$. Then, the measure of Π' is a multiset less than the measure of Π:

LEMMA: `reduce-pattern-property`
`pattern-p`(Π_2) $\wedge$ $w \in \Sigma^*$ $\wedge$ $w \in \mathcal{S}(\Pi_2)$ $\wedge$ $\Pi_1 \in$ `reduce-pattern`(w,Π_2)
$\rightarrow$ `measure-pattern`(Π_1) $<_{\varepsilon_0,\mathcal{M}}$ `measure-pattern`(Π_2)

Finally, as a direct consequence of the previous lemma, if $w \in \mathcal{S}(\mathcal{P})$, then the measure of `reduce-pattern-list`(w,$\mathcal{P}$) is smaller than the measure of $\mathcal{P}$ with respect to $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$:

LEMMA: `reduce-pattern-list-property`
$w \in \Sigma^* \wedge$ `pattern-list-p`($\mathcal{P}$) $\wedge$ $w \in \mathcal{S}(\mathcal{P})$
$\rightarrow$ `pattern-list-measure`(`reduce-pattern-list`(w,$\mathcal{P}$))
$<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$ `pattern-list-measure`($\mathcal{P}$)

Now, we prove that the reduction process only removes strings bigger than some string in the sequence with respect to which the reduction is made. The following lemma establishes this property. If $u \in \mathcal{S}(\mathcal{P})$ then u is still in the set of strings represented by the set of patterns obtained after reducing $\mathcal{P}$ with respect to a given finite sequence of strings $\overline{w}$ ($\overline{w} \in \overline{\Sigma^*} \equiv$ `sigma-*-seq-p`($\overline{w}$)), provided that u is not bigger than any of the strings of $\overline{w}$ (this condition is checked by the function `exists-sigma-*-<=`, whose definition we omit here):

LEMMA: `exists-pattern-reduce-pattern-sequence-list`
($u \in \Sigma^*$ $\wedge$ `sigma-*-seq-p`($\overline{w}$) $\wedge$ `pattern-list-p`($\mathcal{P}$)
$\wedge$ $u \in \mathcal{S}(\mathcal{P})$ $\wedge$ $\neg$`exists-sigma-*-<=`($\overline{w}$,u))
$\rightarrow$ $u \in \mathcal{S}$(`reduce-pattern-sequence-list`($\overline{w}$,$\mathcal{P}$))

In addition, every string is in the initial set of patterns $\{\langle - \rangle\}$:

LEMMA: `initial-pattern-exists-pattern`
$w \in \Sigma^* \rightarrow w \in \mathcal{S}(\texttt{list}(\texttt{initial-pattern}()))$

As a consequence of the above two lemmas, if f_k is not bigger than any of $f_0 \ldots f_{k-1}$ (that is, the recursive case in the definition of `higman-indices`), then if $\mathcal{P}$ is the set of patterns generated in the k-th step, $w \in \mathcal{S}(\mathcal{P})$. So we can use the lemma `reduce-pattern-list-property` to conclude that the measure of the argument in the recursive call in `higman-indices` decreases with respect to $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$. That is, we have the following theorem:

THEOREM: `higman-indices-termination-property`
$k \in \mathbb{N} \wedge \neg\texttt{get-sigma-*-<=-f}(k,\texttt{f}(k))$
$\rightarrow \texttt{higman-indices-measure}(k+1) <_{\varepsilon_0,\mathcal{M}_\mathcal{M}} \texttt{higman-indices-measure}(k)$

This is exactly the proof obligation generated to show the termination of the function `higman-indices`. Thus, its definition is admitted in ACL2 and then the theorem `higman-lemma` presented in subsection 2 is easily proved.

3 Concluding Remarks

The complete ACL2 files with definitions and theorems are available on the Web at `http://www.cs.us.es/~fmartin/acl2/higman/`. To quantify the proof effort, it was done in about 3 weeks of partial time work and only 43 definitions and 76 lemmas (with 17 non trivial proof hints explicitly given) were needed, which gives an idea of the degree of automation of the proof. The development benefits from the previously developed `multiset` book, which provides a proof of well-foundedness of the multiset relation induced by a well-founded relation. It is worth emphasizing the reuse of the `defmul` tool for generating multiset well-founded relations in ACL2 (see [13] for more uses of this tool).

Our proof is slightly different from the one presented in [10]. For example, our construction of the order used to prove the termination of `higman-indices` is more concise. Another important point is the level of detail that we must have in the formalization; this reveals important properties needed in the development of the proof that are not mentioned in [10]. For example, to prove the lemma `exists-pattern-reduce-pattern-sequence-list` we need a stability property about $\preceq^*$: $w_1 \preceq^* w_2 \wedge w_3 \preceq^* w_4 \rightarrow w_1 w_3 \preceq^* w_2 w_4$. The proof of this property is not trivial in our formalization.

There are several constructive mechanizations of Higman's lemma in the literature, for example [2] in the Isabelle system (the author also has the same work done in the COQ system), based on Coquand's constructivization [3] of Nash-William classical proof [11]; and [14] in the MINLOG system. These works are related to program extraction from proofs, whereas our work starts with a program solving the problem and then we prove its properties. This different approach results in a more concise code: our program has 18 lines of LISP code

whereas in [2] the program has 70 lines of ML code; and a more simple result: our program returns the first elements in the sequence such that $w_i \preceq^* w_j$, but this is not the case in [2,14]. On the other hand these works are more restricted than the presented here: in [2] the set Σ has only two values and the well-quasi-order relation is equality; this is not the case in [14] where the proof developed is more general, but the MINLOG formalization is restricted to a finite alphabet.

Acknowledgment

We are grateful to Matt Kauffman and the referees for their helpful comments and suggestions on this paper. Thanks to Matt we have a better understanding of the ACL2 behavior.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. S. Berghofer. *A constructive proof of Higman's lemma in Isabelle.* In *Types for Proofs and Programs, TYPES'04.* LNCS, 3085: 66-82. Springer Verlag, 2004.
3. T. Coquand and D. Fridlender. *A proof of Higman's lemma by structural induction.* Unpublished draft, available at http://www.brics.dk/ daniel/texts/open.ps, 1993.
4. N. Dershowitz and Z. Manna. Proving Termination with Multiset Orderings. *Communications of the ACM* 22(8):465–476, 1979.
5. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
6. M. Kaufmann and J S. Moore. *ACL2 Version 2.9*, 2005.
Homepage: `http://www.cs.utexas.edu/users/moore/acl2/`
7. M. Kaufmann and J S. Moore. *Structured Theory Development for a Mechanized Logic.* Journal of Automated Reasoning, 26(2): 161–203, 2001.
8. F.J. Martín–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina. A Formal Proof of Dickson's Lemma in ACL2. In *Proceedings of LPAR'03.* LNAI, 2850: 49-58. Springer Verlag, 2003.
9. I. Medina–Bulo, F. Palomo, J.A. Alonso, J.L Ruiz–Reina. Verified Computer Algebra in ACL2 (Gröbner basis Computation) In *Proceedings of AISC'04.* LNAI, 3249: 171-184. Springer Verlag, 2004.
10. C. Murthy, J.R. Russell. A Constructive Proof of Higman's Lemma. In *Fifth annual IEEE Symposium on Logic in Computer Science*, pp 257-267, 1990.
11. C. Nash–Williams. *On well-quasi-ordering finite trees.* Proceedings of the Cambridge Philosophical Society, 59:833–835, 1963.
12. H. Perdry. Strong noetherianity: a new constructive proof of Hilbert's basis theorem. Available at `http://perdry.free.fr/StrongNoetherianity.ps`
13. J.L. Ruiz–Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martín–Mateos. Termination in ACL2 Using Multiset Relations. In *Thirty Five Years of Automating Mathematics.* Kluwer Academic Publisher, 2003.
14. M. Seisenberger. *An Inductive Version of Nash-Williams' Minimal-Bad-Sequence Argument for Higman's Lemma.* In *Types for Proofs and Programs, TYPES'00.* LNCS, 2277: 233-242. Springer Verlag, 2001.
15. S.G. Simpson. Ordinal numbers and the Hilbert basis theorem. *Journal of Symbolic Logic* 53(3): 961–974, 1988.

Proof Pearl: Dijkstra's Shortest Path Algorithm Verified with ACL2

J Strother Moore[1] and Qiang Zhang[2]

[1] Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, USA
moore@cs.utexas.edu
http://cs.utexas.edu/users/moore
[2] Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, USA
qzhang@cs.utexas.edu
http://cs.utexas.edu/users/qzhang

Abstract. We briefly describe a mechanically checked proof of Dijkstra's shortest path algorithm for finite directed graphs with nonnegative edge lengths. The algorithm and proof are formalized in ACL2.

1 Introduction and Related Work

Dijkstra's shortest path algorithm [3,4] finds the shortest paths between vertices of a finite directed graph with nonnegative edge lengths. This paper formalizes that claim in ACL2 [8] and briefly describes a mechanically checked proof of it.

ACL2 is a Boyer-Moore style theorem prover by Kaufmann and Moore that supports a first-order logic based on recursively defined functions and inductively constructed objects. The syntax is that of Lisp, which we use (and paraphrase) in this paper – contrary to the TPHOLS tradition – since "proof pearls" are meant to show how certain theorems are proved in certain systems. The ACL2 syntax does not include quantifiers, but the logic provides a means of introducing "Skolem functions" providing full first-order power at the expense of executability. This facility is crucial to the proof described here.

We represent graphs in ACL2 with a list data structure called an association list, explained below. We define the function `dijkstra-shortest-path` to implement the algorithm. It takes two vertices, a and b, and a graph as input and it returns a value, say ρ. We prove that ρ is either `nil` or a path in the graph from a to b, and that no path in the graph from a to b is shorter than ρ. In our formalization, the non-path `nil` has "infinite" length and all finite paths are shorter. Hence, our theorem ensures that if ρ is `nil`, there is no path from `a` to `b`.

Despite the age and classic nature of the algorithm, there is relatively little work on the correctness of Dijkstra's algorithm in the mechanical theorem proving literature. As far as we are aware, the first mechanically checked proof

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 373–384, 2005.

of the correctness of the algorithm was done in Mizar by Jing-Chao Chen [2] in a paper submitted March 17, 2003. For the record, the first ACL2 proof was completed in September, 2003. Our proof requires significant user guidance, but our script is about one third the size of the Mizar proof script.[1] In addition, the Mizar article draws upon notation and results in 22 other Mizar articles concerning properties of sets, functions, arithmetic, chains, and graphs. The ACL2 proof uses no external definitions or theorems – everything is done from ACL2's basic "bootstrap theory." We compare the two proofs in more detail in the concluding section of this paper. The Mizar model of the algorithm is quite similar to ours. In fact, the article states that it was "extremely difficult" to use existing Mizar models of computing machines to formalize the algorithm and instead "we adopt functions in the Mizar library, which seem to be pseudo-codes, and are similar to those in the functional programming, e.g., Lisp." The invariant maintained by the Mizar algorithm is essentially the same as ours, but is expressed in terms of the subgraph "induced" on a larger graph by a subset of the nodes, while our invariant is phrased in terms of paths through the original graph that are "confined" to that subset of nodes.

Joe Hurd [6] formalized and proved the reachability property of Dijkstra's algorithm in HOL. A similar algorithm, Floyd's all-pairs shortest path algorithm, was formalized and proved correct in Coq by Eric Fleury in July, 1990 [5] (unpublished manuscript). In February, 1998, Christine Paulin and Jean-Christophe Filliâtre proved Floyd's algorithm in Coq [10].

In ACL2, Moore did the first proof of Dijkstra's algorithm in September, 2003. He then challenged Zhang, then a relatively new ACL2 user, to do it as an exercise, without seeing Moore's proof. Zhang completed his first proof in December, 2003, with some guidance from Moore. Then Zhang cleaned up his proof, removing many user-supplied proof hints in the process. The proof described here is Zhang's second proof.

The rest of this paper is organized as follows. In the next section we describe the formalization of the algorithm in ACL2. In the subsequent sections we give our specification, the main invariant, a sketch of the proof, and a typical user-supplied hint. In Section 7 we give some statistics about it. The complete script of our work is available online at `http://www.cs.utexas.edu/users/qzhang/shortest-path/index.html`.

[1] The Mizar script, not counting the scripts it references, contains about 132,000 characters, about 30,000 lexical tokens, and about 3,480 lines. The ACL2 script contains about 35,500 characters, about 8,400 lexical tokens, and about 1,200 lines. Parentheses are counted as lexical tokens in both scripts. Comments are not counted. At a higher level of abstraction, the proofs are about the same size. The Mizar script proves 125 lemmas, of which 55 are called "Theorems" (the rest are supporting lemmas). The ACL2 script contains 125 lemmas, of which 57 mention concepts specific to the algorithm, specification, or invariant. The rest are generic. We present more statistics about the ACL2 proof in the concluding section of this paper.

2 Formalization

The ACL2 language is a subset of Common Lisp. We use Lisp syntax. Suppose `neighbors` is a function of two arguments. Then we write `(neighbors u g)` to denote the application of that function to `u` and `g`. That is, we write `(neighbors u g)` where $neighbors(u, g)$ would be written in traditional notation. Lisp conventions make the capitalization of symbols irrelevant (for the particular symbols used in this paper). Thus `neighbors`, `Neighbors`, and `NEIGHBORS` all denote the same symbol. We use lowercase consistently, but capitalize symbols when they occur as the first word of a sentence.

In ACL2, ordered pairs are called conses and are constructed by the function `cons`. The left component of a `cons` is accessed with the function `car` and the right component is accessed with `cdr`. Conses are used to represent lists. The `car` of such a cons is the first element in the list and the `cdr` is the list containing the remaining elements. A list is said to be a *true-list* if its `cdr`-chain is terminated with `nil` rather than some other atom.

An *association list* (or *alist*) is a true-list in which the elements are pairs in which the `car` is said to be *associated with* or *bound to* the `cdr`. It is easy to write the function that looks up the value of a key in an alist, by recurring down the `cdr`-chain to the first pair that binds the key in question. It is also easy to write the function that copies an alist inserting a new binding for given key.

We use alists extensively in this work. A directed graph is an alist associating vertices with edge lists. An *edge list* is an association list associating vertices to nonnegative rationals called *edge lengths*.

The function `graphp` checks that an object is of the shape just described. If the edge list associated with some vertex `u` in a graph `g` binds `v` to `w`, then it means there is an edge in `g` from `u` to `v` with edge length `w`, i.e., `(edge-len u v g)` is `w`. The function `all-nodes` collects a duplication-free true-list ("set") of all vertices mentioned in a graph. `(Neighbors u g)` returns the set of all vertices reachable from vertex `u` via one edge in `g`.

Of course, there are other representations of graphs. The particular one chosen here is unimportant once we have defined and proved the basic properties of `graphp`, `all-nodes`, `neighbors`, `edge-len`, etc. Graphs are accessed entirely via these generic concepts (and no graph is constructed by the algorithm). We thus could have merely constrained these functions to be in the appropriate relationships and conducted our proof without a concrete representation of graphs. We prefer defining such concepts on a concrete representation to establish that functions satisfying all those constraints indeed exist. In fact, ACL2 forces us to provide function definitions "witnessing" any such collection of constraints to establish consistency. In addition, an executable model of the algorithm allows testing, which is particularly helpful when one is trying to formulate lemmas and invariants.

Some readers may prefer a more abstract concrete representation, e.g., as functions. But ACL2 is a first-order language and we do not have functions as objects. However, alists – lists of ordered pairs – are the standard Lisp representation of tables and other finite function objects. When we say "`v` is bound to

`w` in alist `f`" then one may think of `f` as a finite function, `v` as an element of its domain, and `f(v)` = `w`. If one regards alists as finite function objects, then graphs are finite functions from vertices to finite functions from vertices (neighbors) to nonnegative rationals (edge lengths).

A path `p` in a graph `g` is a non-empty list of vertices with the property that successive elements of `p` are linked by an edge in the graph `g`. `Path-len` returns the sum of the edge lengths of the edges in a path.

In ACL2 it is common to use the atom `nil` for a variety of extended meanings. It is used both as the terminal marker in true-lists and as the "false" truth-value. We also use it as "infinity" in our system of lengths. That is, we define a strict ordering `lt` ("less than") and its weaker counterpart `lte` ("less than or equal") so that `nil` dominates all rational lengths. We also use `nil` to denote a non-existent path; that is, if asked to find a path between two vertices where no such path exists, we will return `nil`. We define `path-len` to return `nil` (infinity) on the non-path `nil`. In an abuse of the strictness implied by the word "shorter," we define `(shorterp p1 p2 g)` to be `(lte (path-len p1 g) (path-len p2 g))`.[2]

The core of Dijkstra's shortest path algorithm is an iterative procedure, here called `dsp` (for Dijkstra's shortest path), that computes a *path table*. In our work, path tables are association lists (finite functions) from vertices to paths. All the paths start at the same source vertex. Suppose the source vertex is `a`. Then if `u` is paired with path `p` in the path table, then `p` is a path from `a` to `u`. Other important invariants on the path table are discussed later. We use the variable symbol `pt` to denote the path table. We define `(path u pt)` to return the path associated with `u` in `pt` (or `nil` if no path is associated with `u`) and we define `(d u pt g)` to return its length, `(path-len (path u pt) g)`.

The `dsp` function is defined recursively as shown below.

```
(defun dsp (ts pt g)
  (cond ((endp ts) pt)
        (t (let ((u (choose-next ts pt g)))
             (dsp (del u ts)
                  (reassign u (neighbors u g) pt g)
                  g)))))
```

Here, `dsp` is the name of the function and it takes three arguments: `ts` (the "temporary set" of nodes not yet visited, `pt` (the path table), and `g` (the graph to be explored). We can interpret this recursive function definition operationally as follows. To compute `(dsp ts pt g)`, ask whether `ts` is empty. If so, return `pt`. Otherwise, let `u` be the value of the `choose-next` expression and call `dsp` recursively on `(del u ts)`, the `reassign` expression, and `g`.

From the traditional description of the iterative core of the algorithm the reader should be able to infer the definitions of the functions used above.

Repeat until `ts` is empty:

Choose `u` in `ts` such that `(d u pt g)` is minimal.

[2] Some authors write "(weakly) shorter."

For each edge from `u` to some neighbor `v` with edge length `w`, if `(d v pt g)` > `(d u pt g)` + `w`, then modify `pt` so that the path associated with `v` is the current path to `u` in `pt`, extended onward to `v`, `(append (path u pt) (list v))`.
Delete `u` from `ts`.

We then define Dijkstra's algorithm as

```
(defun dijkstra-shortest-path (a b g)
  (let ((pt (dsp (all-nodes g) (list (cons a (list a))) g)))
    (path b pt)))
```

which may be described as:

Let `pt` be the final path table computed by `dsp` starting from an initial `ts` containing all the nodes of the graph and an initial path table pairing the source vertex `a`, with the singleton path that starts and ends at `a`.
Return the path associated with `b` in the final path table.

Given that ACL2 is a functional programming language, this algorithm may be executed on concrete input, though as coded here it is not very efficient. Much more efficient implementations are possible in ACL2, e.g., using ACL2's single-threaded objects [1] (which are data structures that may be modified destructively but under syntactic restrictions that ensure conformance to the applicative semantics) and the `MBE` feature (which permits the replacement of one ACL2 code fragment by another provided they are provably equivalent in the given context). These features could be used to implement the array-based binary trees commonly employed to represent the path table efficiently; the key step would be a commuting diagram relating the "accessor" function, `path`, which recovers the path associated with a given vertex in the path table, to the "updater" function, `reassign`, for the two different concrete representations of path tables.

To our knowledge, no ACL2 proof of such an implementation has been carried out. But correctness proofs by Sumners and Ray for an *in situ* ACL2 quicksort [12], Sumners for an ACL2 BDD package that operates at 60% of the speed of CUDD [11], and Greve and Wilding for an ACL2 graph path finding algorithm that executes at speeds near those of a C implementation [7] are evidence that moving from this implementation to an efficient one in ACL2 is a well-trodden path. The main obstacle is proving the correctness of some ACL2 function implementing the algorithm in question.

3 Specification

Our specification of the algorithm is as follows. Suppose `a` and `b` are nodes in graph `g`. Let ρ (`rho` below) be the output of Dijkstra's algorithm on `a`, `b`, and `g`. Then ρ is either `nil` or a path in `g` from `a` to `b`, and ρ is a (weakly) shortest path from `a` to `b` in `g`. Note that if ρ is `nil` the claim that it is nevertheless the shortest path from `a` to `b` is equivalent to the claim that there is no such path, since any true path from `a` to `b` is shorter than the infinite `path-len` of `nil`. Formally, in ACL2, we write this as follows.

```
(defthm main-theorem
  (implies (and (nodep a g)
                (nodep b g)
                (graphp g))
           (let ((rho (dijkstra-shortest-path a b g)))
             (and (or (null rho)
                      (pathp-from-to rho a b g))
                  (shortest-pathp a b rho g)))))
```

To formalize the notion that a path is a (weakly) shortest path we define `(shortest-pathp a b p g)` so that it is true if and only if for every path, `path`, from `a` to `b` in `g`, `p` is (weakly) shorter than `path`. We could "fake" this quantification with a recursive function that checks all possible paths, if there were a finite number of them. But in general there may be an infinite number of (non-simple) paths to a given node. ACL2 does not provide quantifiers *per se*. But it does provide a facility, `defchoose` [9], like Hilbert's ϵ, by which one can introduce a function to return an object satisfying a given formula, if such an object exists.

Therefore, to define `shortest-pathp` we first use `defchoose` to introduce a witness, `(shortest-pathp-witness a b p g)` with the property that it is a path in `g` from `a` to `b` and is shorter than `p`, if such an object exists. Then we define `shortest-pathp` so that it is true of `p` precisely if the witness fails to be a path from `a` to `b` that is shorter than `p`. In ACL2, this entire development is wrapped up in a macro called `defun-sk` (for "define Skolemized function").

```
(defun-sk shortest-pathp (a b p g)
 (forall path
         (implies (pathp-from-to path a b g)
                  (shorterp p path g))))
```

The macro expands to an appropriate use of `defchoose` for the witness expression `(shortest-pathp-witness a b p g)` followed by an appropriately encapsulated definition of `shortest-path`. This method of introducing quantified concepts in ACL2 differs from the method in Nqthm, where Skolemization was supported directly.

Such Skolem functions are not executable: even when the arguments are known constants, ACL2 cannot reduce a call of `shortest-pathp-witness` to a constant. This does not trouble us because these functions are used in the specification and proof, but not in the path-finding algorithm itself.

The witness function is used extensively in a series of hand-written hints used to carry out the most delicate arguments in the correctness proof. In particular, to show that a just-constructed path is a shortest one, we suppose it is not, use the witness to obtain an allegedly shorter one, and then derive a contradiction. But while the various case splits and constructions used to conduct these arguments are the messiest part of the proof, the real crux of the proof is identifying and formalizing the invariant maintained by the algorithm.

4 The Invariant

The mechanical proof is mainly concerned with establishing an invariant on the temporary set, `ts` and the path table, `pt`, of `dsp`. The invariant also takes the starting vertex, `a`, and the graph, `g`.

Several concepts are used repeatedly in defining the invariant. One is the notion of the "final set," usually represented here by the variable `fs` and equal to `(comp-set ts (all-nodes g))`, the complement of the temporary set (with respect to the set of all nodes of the graph). Another is the idea of a path `p` being *confined* to `fs`, which means that every node in `p` except the last is a member of `fs`. We define the concept recursively.

```
(defun confinedp (p fs)
  (if (endp p) t
    (if (endp (cdr p)) t
      (and (memp (car p) fs)
           (confinedp (cdr p) fs)))))
```

A third important concept is that of `p` being a *shortest confined path*, meaning it is shorter than any path from `a` to `b` that is confined to `fs`. We need universal quantification (`defun-sk`) to formalize this.

```
(defun-sk shortest-confined-pathp (a b p fs g)
  (forall path (implies (and (pathp-from-to path a b g)
                             (confinedp path fs))
                        (shorterp p path g))))
```

We define the invariant as follows:

```
(defun invp (ts pt g a)
    (let ((fs (comp-set ts (all-nodes g))))
      (and (ts-propertyp a ts fs pt g)
           (fs-propertyp a fs fs pt g)
           (pt-propertyp a pt g))))
```

The invariant has three conjuncts, one each about the temporary set, the final set, and the path table, although this partitioning is somewhat artificial since all involve `fs` and `pt` to some extent.

We define `ts-propertyp` recursively to check that for every node in the temporary set, the path to that node in the path table is a shortest confined path to that node and the path is itself confined.

```
(defun ts-propertyp (a ts fs pt g)
  (if (endp ts) t
    (and (shortest-confined-pathp a (car ts)
                                  (path (car ts) pt)
                                  fs g)
         (confinedp (path (car ts) pt) fs)
         (ts-propertyp a (cdr ts) fs pt g))))
```

We define `fs-propertyp` recursively in a very similar fashion, except it checks that for every node in the final set, the path assigned to that node in the path table is a shortest path to that node and is confined.

```
(defun fs-propertyp (a fs fs0 pt g)
  (if (endp fs) t
    (and (shortest-pathp a (car fs) (path (car fs) pt) g)
         (confinedp (path (car fs) pt) fs0)
         (fs-propertyp a (cdr fs) fs0 pt g))))
```

Finally, we define `pt-propertyp` to check that for every entry in the path table is either `nil` or a path from `a` to the node with which it is associated in the table.

```
(defun pt-propertyp (a pt g)
  (if (endp pt) t
    (and (or (null (cdar pt))
             (pathp-from-to (cdar pt) a (caar pt) g))
         (pt-propertyp a (cdr pt) g))))
```

5 Mechanical Proof

The proof breaks down into two main lemmas. The first is that the invariant holds initially.

```
(defthm invp-0
  (implies (nodep a g)
           (invp (all-nodes g) (list (cons a (list a))) g a)))
```

The second is that the invariant holds as `dsp` recurs.

```
(defthm invp-choose-next
  (implies (and (invp ts pt g a)
                (my-subsetp ts (all-nodes g))
                (graphp g)
                (consp ts)
                (setp ts)
                (nodep a g)
                (equal (path a pt) (list a)))
           (let ((u (choose-next ts pt g)))
             (invp (del u ts)
                   (reassign u (neighbors u g) pt g)
                   g a)))
  :hints ...)
```

From these two, it is straightforward to prove

```
(defthm invp-last
  (implies (and (nodep a g)
                (graphp g))
```

```
        (invp nil
              (dsp (all-nodes g)
                   (list (cons a (list a)))
                   g)
              g a)))
```

and `main-theorem` follows without much more work.

6 Hints

The hardest part of the proof is, of course, the proof of `invp-choose-next`. We present only one of the major cases. `Dsp` uses `choose-next` to choose a vertex, `u`, in `ts` whose associated path in `pt` is of minimal length. Why is this path the shortest path to that vertex? Here is the lemma that states that it is.

```
(defthm choose-next-shortest
  (implies (and (graphp g)
                (consp ts)
                (my-subsetp ts (all-nodes g))
                (invp ts pt g a))
           (shortest-pathp a
                           (choose-next ts pt g)
                           (path (choose-next ts pt g) pt)
                           g))
  :hints ...)
```

ACL2 cannot prove this without help. Help is given by the user in the form of hints. We first describe the proof and then show the actual hints.

Let the `choose-next` term above be u and let its associated path in `pt` be δ. Let *fs* be the "final set," `(comp-set ts (all-nodes g))`. We know, from the `invp` hypothesis, that δ is the shortest path to u that is confined to *fs*. We wish to show it is the shortest path (confined or not). Suppose it is not. Then there is a shorter path, say σ, to u that is not confined to *fs*, i.e., σ contains a vertex v in `ts`. Let σ' be the initial portion of σ up to and including v. Then σ' is shorter than σ, terminates on a node in `ts`, and is confined to *fs*. But the path in `pt` associated with v is, by `invp`, shorter than σ'. And δ is shorter than that path by the selection criteria in `choose-next`. Hence, δ is shorter than σ.

The actual term for σ above is `(shortest-pathp-witness a` u δ `g)`. And the actual term for σ' is `(find-partial-path` σ *fs*`)`. `Find-partial-path` is a user-defined recursive function that finds the subpath of a path that terminates in the first node outside of *fs*.

Hints in ACL2 are generally coded by listing a series of instantiations of previously proved lemmas. These instances are conjoined to the hypotheses of the goal theorem and then used freely by ACL2. To code the above hint we tell ACL2 not to expand the definitions of `shorterp`, `path` and `pathp` and we provide two instances. The ellipsis in the display above for `choose-next-shortest` is filled in by:

```
(("Goal" :in-theory (disable shorterp path pathp)
  :use ((:instance pathp-partial-path (p σ) (s fs))
        (:instance shorterp-by-partial-and-choose-next
         (u u) (path σ′) (v (car (last σ′)))))))
```

The expression following the symbol `:use` specifies that the theorem prover is to add two lemma instances to the hypotheses of the goal. The first lemma, `pathp-partial-path`, instantiated above says that `find-partial-path` constructs a confined path to its last node. The given substitution replaces the variable symbol `p` in the lemma by σ and the variable `s` by *fs*. The second lemma says that if the path to `u` in `pt` is shorter than the path to `v` in `pt`, and `ts-propertyp` holds, and `path` is a confined path to `v`, then the path to `u` is shorter than `path`.

7 Some Details and Statistics

The entire ACL2 proof script contains 39 `defun`s and 125 `defthm`s. The `defthm`s can be broken into to two broad categories: elementary lemmas about the basic ideas and "custom" lemmas for this particular proof. We classified as "custom" any lemma mentioning `choose-next`, `reassign`, `ts-propertyp`, `fs-propertyp`, `pt-propertyp`, `invp`, `dsp`, or `dijkstra-shortest-path`.

There are 68 elementary lemmas about finite set theory, the notions of shorter and shortest path, elementary path properties (including that of being confined) and manipulation (including the notion of finding a confined subpath), and structural properties of association lists, paths, tables, and graphs. Here are a few.

```
(defthm comp-set-id
  (equal (comp-set s s) nil))
(defthm neighbor-implies-nodep
  (implies (memp v (neighbors u g))
           (memp v (all-nodes g))))
(defthm shortest-pathp-corollary
  (implies (and (shortest-pathp a b p g)
                (pathp-from-to path a b g))
           (shorterp p path g)))
(defthm confinedp-append
  (implies (and (confinedp p s)
                (memp (car (last p)) s))
           (confinedp (append p (list v)) s)))
(defthm path-len-append
  (implies (pathp p g)
           (equal (path-len (append p (list v)) g)
                  (plus (path-len p g)
                        (edge-len (car (last p)) v g)))))
```

All are used by ACL2 as conditional rewrite rules. For example, the last theorem is used to rewrite `(path-len (append ...))` to the `plus` expression, provided `(pathp p g)` can be established. (`Plus` is just addition extended to handle `nil` as "infinity.")

There are 57 custom lemmas, including four shown in this paper: `invp-0`, `invp-choose-next`, `invp-last`, and `choose-next-shortest`. Some are easy to prove lemmas that "explain" the fact that functions like `ts-propertyp` are recursively defined quantifiers:

```
(defthm ts-propertyp-prop-lemma1
  (implies (and (ts-propertyp a ts fs pt g)
                (memp v ts))
           (and (shortest-confined-pathp a v (path v pt) fs g)
                (confinedp (path v pt) fs))))
```

In all, we had to give 51 hints. About 30 of these were hints only to disable (i.e., avoid using) certain definitions or theorems. Twenty-three times we had to instruct the theorem prover to `:use` instances of certain theorems, as illustrated above, and a total of 31 instances were mentioned in the script. The vast majority of the hints were used in the custom theorems: 37 of the 51 hints, 19 of the 23 `:use` hints for 28 of the 31 instances.

The proof takes about 67 seconds on a 2.4 GHz Intel Xeon^{TM} running ACL2 Version 2.9 compiled under GNU Common Lisp.

References

1. R. S. Boyer and J S. Moore. Single-threaded objects in ACL2. In *PADL 2002*, pages 9–27, Heidelberg, 2002. Springer-Verlag LNCS 2257. `http://www.cs.utexas.edu/users/moore/publications/stobj/main.ps.gz`.
2. Jing-Chao Chen. Dijkstra's shortest path algorithm. *Journal of Formalized Mathematics*, vol. 15, 2003.
3. E. W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math. 1*, pages 269–271, 1959.
4. Shimon Even. *Graph Algorithms*, chapter 1. Computer Science Press, Inc., 1979.
5. Eric Fleury. Implantation des algorithmes de Floyd et de Dijkstra dans le Calcul des Constructions. Rapport de Stage, July 1990.
6. M. Gordon, J. Hurd, and K. Slind. Executing the formal semantics of the Accellera property specification language by mechanized theorem proving. In D. Geist, editor, *Proceedings of CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 200–215. Springer Verlag, 2003.
7. D. Greve and M. Wilding. Using mbe to speed a verified graph pathfinder. In *ACL2 Workshop 2003*, Boulder, Colorado, July 2003. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`.
8. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
9. M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.
10. C. Paulin and J. C. Filliâtre. `http://pauillac.inria.fr/cdrom/www/coq/contribs/floyd.html`.

11. R. Sumners. Correctness proof of a BDD manager in the context of satisfiability checking. In *Proceedings of ACL2 Workshop 2000*. Department of Computer Sciences, Technical Report TR-00-29, November 2000. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2000/final/sumners2/paper.ps`.
12. R. Sumners and S. Ray. Verification of an in-place quicksort in ACL2. In *Proceedings of the ACL2 Workshop, 2002*. `http://www.cs.utexas.edu/~moore/acl2/workshop-2002`, Grenoble, April 2002.

Proof Pearl: Defining Functions over Finite Sets

Tobias Nipkow[1] and Lawrence C. Paulson[2]

[1] Institut für Informatik, Technische Universität München, Germany
[2] Computer Laboratory, University of Cambridge, England

Abstract. Structural recursion over sets is meaningful only if the result is independent of the order in which the set's elements are enumerated. This paper outlines a theory of function definition for finite sets, based on the fold functionals often used with lists. The fold functional is introduced as a relation, which is then shown to denote a function under certain conditions. Applications include summation and maximum. The theory has been formalized using Isabelle/HOL.

1 Introduction

Finite sets have many applications in interactive proof. Lists are more commonly used, but are a poor substitute for sets: if the order of the list elements really doesn't matter, then the use of lists will introduce meaningless distinctions and pointless complications. In general, of course, sets do not have to be finite. Theorem provers based on higher order logic can easily reason about infinite sets. Finite sets are appropriate for modelling computational phenomena such as message buffering, where the order and possible repetition of messages must be ignored. Finite sets support operations—such as summation, maximum and minimum, cardinality—that are meaningful for infinite sets only in the context of the calculus or other advanced methods.

Fold functionals are a convenient means of defining such operations. For lists, fold functionals are well known [8], but for finite sets, they are problematical. The problem is obvious: fold functionals seem to presuppose an ordering of the elements, when by definition a set is unordered. The solution is to define a fold functional that allows the definition of operations where the order of elements is irrelevant.

- Summations are unaffected by the order in which elements are added.
- The maximum and minimum do not depend on the order in which elements appear.
- Cardinality is unaffected by the order in which elements are counted.

This paper has several objectives. We describe how to the formalize fold functionals for finite sets. While presenting our approach, which we believe provides maximum flexibility for minimal effort, we identify alternatives and outline their merits. No mathematical novelties appear here, just a carefully tuned series of definitions. We use Isabelle/HOL [9], but the approach should be applicable

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 385–396, 2005.

to other theorem provers based on higher-order logic or set theory. We also illustrate the technique of proving a function to be well-defined by showing the corresponding relation to be single valued.

We pay particular attention to the algebraic properties required for iterating a function over a set. It turns out that there are two distinct cases: commutative monoids with a unit (useful for defining summation) and ordered structures (useful for defining minimum). Both require distinct fold functionals and their own theory. In the development of these theories we demonstrate *locales*, a lesser-known Isabelle feature.

As we go along, we compare our approach with the one in HOL4 [7] and PVS [13], both of which provide their own libraries of functions over finite sets. Our basic fold function resembles one formalized for HOL88 by Chou [3].

2 Finite Sets and the Fold Function

We assume there already is a formalization of sets, with standard operations such as comprehension, union and intersection. In higher-order logic, this is trivial by the obvious representation of sets by predicates.

We use standard mathematical notation with a few extensions. Type variables are written $'a$, $'b$, etc., and function types are written using $\Rightarrow$, as in $'a \Rightarrow 'a$. Logical equivalence is denoted by $=$, equality between booleans. The type of sets over a type $'a$ is $'a\ set$. The image of a function over a set, namely $\{f\ x \mid x \in A\}$, is written $f\ `\ A$. Injectivity is written $inj\text{-}on\ f\ A$, which means the function f is injective when restricted to the set A.

We use two description operators, *SOME* and *THE*. Both denote a value that is specified by a formula. *SOME* is Hilbert's ϵ-operator; it does not require the formula to specify the value uniquely, instead choosing one using the axiom of choice. *THE* is a *definite* description: the specified value must be unique.

Most uses of *SOME* that we have seen can be replaced by *THE*. Sometimes, *SOME* is essential or at least leads to shorter definitions. However, introducing a needless dependence on the axiom of choice is inelegant. Certain formal systems are incompatible with choice [5, Remark 4.6].

2.1 Finite Sets

Finiteness we express by an inductive definition. The empty set is finite, and adding one element to a finite set yields a finite set. The inductive definition defines a predicate *finite* characterizing the finite sets:

$$finite\ \emptyset \qquad \frac{finite\ A}{finite\ (\{a\} \cup A)}$$

In Isabelle, we define the function *insert* and abbreviate the conclusion of the second rule as *finite (insert a A)*. This function brings out the analogy between finite sets built by $\emptyset$ and *insert* with lists built by [] and *Cons*. We do not introduce a separate type of finite sets, which might be desirable in systems that offer predicate subtyping.

Isabelle's inductive definition package also generates an induction rule similar to structural induction for lists. Familiar properties are easily proved by induction. For example, the union of finitely many finite sets is itself finite.

Traditionally, a set is finite if it can be put into one-to-one correspondence with the set of natural numbers less than some n. Another traditional definition says A is finite if every injection from A to A is also a surjection. Compared with the inductive definition of finiteness, which allows the standard results to be proved easily, such definitions are inconvenient. Their only advantage is that they do not presuppose the concept of inductive definition.

If we did not start from an existing type of sets which includes the infinite ones, we could define the type of finite sets as a quotient type of a free algebra [11]. There are two standard approaches.

- empty set, singleton set and union, modulo associativity, commutativity and idempotency of union
- empty set and insert, modulo *left-commutativity* and *left-idempotency*:

$$\begin{gathered} \mathit{insert}\ a\ (\mathit{insert}\ b\ A) = \mathit{insert}\ b\ (\mathit{insert}\ a\ A) \\ \mathit{insert}\ a\ (\mathit{insert}\ a\ A) = \mathit{insert}\ a\ A \end{gathered}$$

Below we will refer to them as the $(\emptyset,\{-\},\cup)$-algebra and the $(\emptyset,\mathit{insert})$-algebra.

2.2 Which Fold Function?

Our main interest is defining functions over finite sets by recursion. For lists, this is trivial, but for sets, we must ensure that the order of the elements is irrelevant. The cardinality of a set is a key function that might be defined recursively, though it turns out to be essential in the development of recursion in the first place.

We seek a function *fold* of type $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ \mathit{set} \Rightarrow 'a$. It should satisfy the equation

$$\mathit{fold}\ f\ g\ z\ \{x_1, ..., x_n\} = f\ (g\ x_1)\ (\ldots\ (f\ (g\ x_n)\ z)\ldots)$$

if f is associative and commutative (AC). The function g is applied to each of $\{x_1, ..., x_n\}$, then z is thrown in, and the resulting values —which do *not* have to be distinct—are combined using f. This fold function is also well known from functional data base query languages, for example Machiavelli [10].

If e is a unit element for f, i.e. $f\ x\ e = x$, then *fold* satisfies the recursion equations

$$\begin{gathered} \mathit{fold}\ f\ g\ e\ \emptyset = e \\ \mathit{fold}\ f\ g\ e\ \{a\} = g\ a \\ \mathit{fold}\ f\ g\ e\ (A \cup B) = f\ (\mathit{fold}\ f\ g\ e\ A)\ (\mathit{fold}\ f\ g\ e\ B) \end{gathered}$$

if A and B are finite disjoint sets. Hence *fold* corresponds to the $(\emptyset,\{-\},\cup)$-algebra view of finite sets.

Can the argument g be eliminated? The application of g to the set elements cannot be replaced by a separate use of the image operator because the resulting collection of values must not be regarded as a set. We could combine f and g

into one function, namely $\lambda x\ y.\ f\ (g\ x)\ y$. That approach, followed in HOL4, resembles the treatment of fold for lists, i.e. it takes the $(\emptyset, insert)$-algebra view of finite sets. It can be shown that the two fold functions are interdefinable [2]. Our treatment has the advantage that it only involves standard algebraic properties like associativity and commutativity.

What happens if there is no unit e for f? As an example, consider a naive definition of the minimum of a set of natural numbers: $Min \equiv fold\ min\ id\ 0$ where *min* is the binary minimum (which is AC, but has no unit). The sad consequence of this definition is that *Min* always returns zero. As a matter of fact, *fold* will not allow us to define the minimum of a set over any type that lacks a greatest element: a unit for *min*. Similarly, the maximum of a set can be defined using *fold* only if the type has a least element; for the natural numbers, it is correct to define $Max \equiv fold\ max\ id\ 0$. We treat the case of a missing unit element separately in §4.

2.3 Defining the Fold Function

We do not define *fold* directly. Instead, we inductively define its graph: its input/output relation. After proving that this relation is deterministic, we use it to define *fold*. The relation is a constant *foldSet* of type

$$('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('b\ set \times 'a)\ set.$$

Its inductive definition has two introduction rules. The first, intuitively, states that when applied to the empty set, $fold\ f\ g\ z$ should return z.

$$(\emptyset,\ z) \in foldSet\ f\ g\ z$$

The second rule says that if the "fold" of A can yield y, then the "fold" of $\{x\} \cup A$ can yield $f\ (g\ x)\ y$, for any $x \notin A$.

$$\frac{x \notin A \qquad (A,\ y) \in foldSet\ f\ g\ z}{(\{x\} \cup A,\ f\ (g\ x)\ y) \in foldSet\ f\ g\ z}$$

Proving that the "fold" of A can yield only one value is complicated because the elements of A might be inserted in any order. Our proof (see below) is by induction on the cardinality of A. Once we have proved that *foldSet* corresponds to a function, we can define the "fold" of A to be the unique x determined by $foldSet\ f\ g\ z$:

$$fold\ f\ g\ z\ A \equiv THE\ x.\ (A,\ x) \in foldSet\ f\ g\ z$$

This step requires the unique description operator, but not the axiom of choice.

3 A Fold Function for Finite Sets

All treatments of fold that we are aware of require some notion of finite cardinality. Various definitions are possible.

1. The traditional approach refers to a bijection to an initial segment of the natural numbers: $card\ A \equiv LEAST\ n.\ \exists f.\ A = f\ `\ \{m \mid m < n\}$
2. HOL4 defines cardinality as a relation of pairs $(A,\ n)$. This graph is proved to be deterministic and turned into a function using a description.
3. The approach we adopt below refers implicitly to the traditional definition of finiteness, via the theorem $finite\ A = (\exists n\ f.\ A = f\ `\ \{i \mid i < n\})$.

Alternative 2 above resembles the inductive definition of *fold*. Whichever alternative is chosen, we should only prove enough results about cardinality to allow the definition of *fold*: many lemmas about cardinality are instances of more general lemmas about set summation and can be obtained easily once that concept is defined with the help of *fold*. We come back to this point in §3.5.

As stated in the previous section, the relation *foldSet* is defined inductively. In order to turn *foldSet* into a function, we must show *determinacy*: for each A there is at most one y such that $(A,\ y) \in foldSet\ f\ g\ x$. However, this is not true for arbitrary f: the function must be AC.

3.1 Commutative Monoids

An Isabelle *locale* is essentially a detached proof context, comprising variables and assumptions that are temporarily treated like constants and axioms [1]. When proving theorems within a locale, there is no need to repeat the assumptions nor to discharge them when appealing to other theorems of that locale.

Here, a locale conveniently packages the function f with its associative and commutative laws. Locale *ACf* even gives f an infix syntax $\cdot$ to improve clarity:

locale *ACf* =
  fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\cdot$ *70*)
  assumes *commute*: $x \cdot y = y \cdot x$
  and *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

Some of our proofs require a unit element for f. Locale *ACe* extends *ACf* with such an element, called e:

locale *ACe* = *ACf* +
  fixes $e :: 'a$
  assumes *ident*: $x \cdot e = x$

For clarity, we write $\cdot$ and e only in the context of the corresponding locale. Outside the locale, where the values are unconstrained, we write f and z. In the locale, references to f as a function appear as $(\cdot)$, a notation that emphasizes the connection with the infix syntax $x \cdot y$. An example of this notation in context is $foldSet\ (\cdot)\ g\ x$.

3.2 From Relation *foldSet* to Function *fold*

The determinacy of *foldSet* is easily stated:

If $(A,\ x) \in foldSet\ (\cdot)\ g\ z$ and $(A,\ y) \in foldSet\ (\cdot)\ g\ z$ then $y = x$.

For induction, we insert two conditions, expressing that A has cardinality n:

If $A = h\ `\ \{i \mid i < n\}$ and *inj-on* $h\ \{i \mid i < n\}$ and $(A, x) \in \mathit{foldSet}\ (\cdot)\ g\ z$ and $(A, y) \in \mathit{foldSet}\ (\cdot)\ g\ z$ then $y = x$.

The proof is conducted within locale *ACf*, making available the assumption that $\cdot$ is associative and commutative. We use complete induction on n, obtaining as induction hypothesis that the theorem statement holds for all m less than n.

We next analyse the assumption $(A, y) \in \mathit{foldSet}\ (\cdot)\ g\ z$ by cases on the definition of *foldSet*. If $A = \emptyset$ then both x and y are equal to z. Otherwise A is non-empty. We must consider two potential evaluations of fold:

1. $A = \{b\} \cup B$ and $x = g\ b \cdot u$ and $(B, u) \in \mathit{foldSet}\ (\cdot)\ g\ z$, where $b \notin B$
2. $A = \{c\} \cup C$ and $y = g\ c \cdot v$ and $(C, v) \in \mathit{foldSet}\ (\cdot)\ g\ z$, where $c \notin C$

If $b = c$ then also $B = C$, whence $u = v$ by the induction hypothesis. We thus find that x and y equal $g\ b \cdot u$.

The only remaining case is if $b \neq c$. Here, we define the set $D = B - \{c\}$. Trivially $B = \{c\} \cup D$, but we can also show $C = \{b\} \cup D$ by subtracting c from both sides of the set equation $\{b\} \cup B = \{c\} \cup C$. Because the set D is of smaller cardinality, the induction hypothesis tells us that $(D, w) \in \mathit{foldSet}\ f\ g\ z$ for a unique w. Returning to the two potential evaluations of fold, we find

1. $A = \{b, c\} \cup D$ and $x = g\ b \cdot (g\ c \cdot w)$
2. $A = \{c, b\} \cup D$ and $y = g\ c \cdot (g\ b \cdot w)$.

Associativity and commutativity of $(\cdot)$ imply $x = y$, which completes the proof. □

If a theory of finite cardinality is already available, we can replace the formulas $A = h\ `\ \{i \mid i < n\}$ and *inj-on* $h\ \{i \mid i < n\}$ above by *finite* A. The proof would then require a complete induction on the cardinality of A. The only properies of cardinality needed are the obvious ones: $\mathit{card}\ \emptyset = 0$ and

$$\mathit{card}\ (\{a\} \cup A) = (\mathsf{if}\ a \in A\ \mathsf{then}\ \mathit{card}\ A\ \mathsf{else}\ \mathit{card}\ A + 1).$$

Our experience is that defining cardinality separately requires much work and yields only a small simplification in the formalization of the proof above. Fold will give us cardinality for free.

Now that we have shown that *foldSet* is the graph of a function, we can easily derive the recursion rule for *fold* where A must be finite and $x \notin A$:

$$\mathit{fold}\ (\cdot)\ g\ z\ (\{x\} \cup A) = g\ x \cdot \mathit{fold}\ (\cdot)\ g\ z\ A \tag{1}$$

The base case $\mathit{fold}\ f\ g\ z\ \emptyset = z$ follows directly from the definitions. Once we have these two equations, we can forget about the original definition of *fold*.

3.3 An Alternative: Defining Fold by Choice and Recursion

An alternative definition of *fold* is by well-founded recursion over the cardinality of the set, where $\mathit{pick}\ A \equiv \mathit{SOME}\ a.\ a \in A$ and $\mathit{rest}\ A \equiv A - \mathit{pick}\ A$:

$\mathit{fold}\ f\ g\ x\ A = (\mathsf{if}\ A = \emptyset\ \mathsf{then}\ x\ \mathsf{else}\ \mathit{fold}\ f\ g\ (\mathit{pick}\ A)\ (\mathit{rest}\ A))$

This version, used in PVS and HOL4, is appealingly concise. In PVS the type of *fold* is restricted to finite sets, whereas in HOL the recursion equation is subject to the condition *finite A*. A comparison of the Isabelle and HOL4 proof scripts leading to equation (1) suggests that the use of the axiom of choice yields no reduction in the proof effort. Our approach, which introduces no dependence on the axiom of choice, provides more flexibility at no additional cost.

3.4 Further Properties of Fold

Here is a selection of equations that we have formally proved about the function *fold*. They are all subject to the condition that the sets are finite. Most of the proofs are trivial inductions over the finite set A.

$f\ x\ (fold\ f\ g\ z\ A) = fold\ f\ g\ (f\ x\ z)\ A$
$fold\ f\ g\ (fold\ f\ g\ z\ B)\ A = fold\ f\ g\ (fold\ f\ g\ z\ (A \cap B))\ (A \cup B)$
$fold\ f\ g\ z\ (h\ `\ A) = fold\ f\ (g \circ h)\ z\ A$

The following theorems is proved in locale ACe:

$$fold\ (\cdot)\ g\ e\ A \cdot fold\ (\cdot)\ g\ e\ B = fold\ (\cdot)\ g\ e\ (A \cup B) \cdot fold\ (\cdot)\ g\ e\ (A \cap B) \qquad (2)$$

These samples from the fold library should suffice.

3.5 Applications

Our experience is that users seldom invoke *fold* directly. The great majority of references to *fold* are via functions defined using it. Summation, which is the most important of these, sums a given function over an index set. Also useful is an analogous operator for products. Cardinality is defined in terms of summation.

So far we have implicitly worked in the context of a fixed but arbitrary commutative semigroups or monoid, i.e. the locales ACf or ACe. Now we replace $\cdot$ by addition or multiplication and e by *0* or *1*. This is fine as those are commutative monoids. In Isabelle it means instantiating the locales ACf or ACe and proving their assumptions; the details are discussed elsewhere [1].

Sums Over a Set. We work in a flexible formalization of arithmetic, defined using axiomatic type classes [12]. A general theory of commutative monoids specifies that + is an AC operator with unit element 0. The resulting concept of summation is applicable to integers, rationals, reals, matrices and even multisets.

The sum of the function f over the set A is defined in terms of *fold*.

$setsum\ f\ A \equiv$ *if finite* A *then* $fold\ (+)\ f\ 0\ A$ *else* 0

The sum is defined to be zero if A is infinite; by case analysis on *finite* A, many theorems about *setsum* can be proved without finiteness assumptions. (The analogous if-then definition of *fold* would simplify only a few theorems.) This summation operator inherits the theorems about *fold* shown above.

Products over a set are defined analogously.

Syntactic Sugar. We have attached more conventional concrete syntax to various forms of *setsum*. For a start *setsum* $(\lambda x.\ e)\ A$ can be written (and is always printed) as $\sum x{\in}A.\ e$. Instead of $\sum x{\in}\{x.\ P\}.\ e$ we have the shorter $\sum x \mid P.\ e$. The special form $\sum x{\in}A.\ x$ abbreviates to $\sum A$.

Cardinality of a Finite Set. As remarked above, the summation operator and the theorems proved about it are applicable to all types that belong to the class of commutative monoids. Among these is *nat*, the type of the natural numbers. The cardinality of a finite set can be expressed as a summation:

$card\ A \equiv \sum x{\in}A.\ 1$

Note that the cardinality of an infinite set is zero with this definition. The usual properties of cardinality are instances of those for summations.

4 A Fold Function for Non-empty Sets

Some functions on finite sets, such as *Max*, require their argument to be non-empty. The algebraic reason is that the result type does not have a unit element *e*. This phenomenon is well-known from functional programming with lists, where typically two fold functions are available. We do the same here and define a second fold function *fold1* of type $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a$ such that

$fold1\ (\cdot)\ \{x_1, ..., x_n\} = x_1 \cdot \ldots \cdot x_n$

if $(\cdot)$ is associative and commutative.

Unlike *fold*, the function *fold1* does not apply some g to all x_i. In all our examples, the operator $(\cdot)$ is idempotent, when g can be mapped over the set first. For *fold*, it is easy to show that if $(\cdot)$ is idempotent then g becomes redundant:

$finite\ A \Longrightarrow fold\ (\cdot)\ g\ z\ A = fold\ (\cdot)\ id\ z\ (g\ `\ A)$

The Isabelle definition of *fold1* again avoids the axiom of choice and proceeds as for *fold*: a relation *fold1Set* is defined inductively. We avoid recursion and reduce *fold1Set* to *foldSet*.

$$\frac{(A, x) \in foldSet\ f\ id\ a \qquad a \notin A}{(\{a\} \cup A, x) \in fold1Set\ f}$$

Again, our plan is to convert this relation into a function:

$fold1\ f\ A \equiv THE\ x.\ (A, x) \in fold1Set\ f$

A surprise: this does *not* require proving determinacy of *fold1Set*! The equation for the base case is easy to show: $fold1\ f\ \{a\} = a$. Harder to derive is the recursion rule, where A must be finite and non-empty and $a \notin A$:

$$fold1\ (\cdot)\ (\{a\} \cup A) = a \cdot fold1\ (\cdot)\ A \tag{3}$$

Our proof requires two lemmas to allow us to change *foldSet*'s third argument. Both lemmas are proved by induction on the derivation of their first premise.

$$\frac{(A,\ y) \in \mathit{foldSet}\ (\cdot)\ \mathit{id}\ b \qquad b \notin A}{(\{b\} \cup A,\ z \cdot y) \in \mathit{foldSet}\ (\cdot)\ \mathit{id}\ z}$$

$$\frac{(A,\ x) \in \mathit{foldSet}\ (\cdot)\ \mathit{id}\ b \qquad a \in A \qquad b \notin A}{(\{b\} \cup (A - \{a\}),\ x) \in \mathit{foldSet}\ (\cdot)\ \mathit{id}\ a}$$

From these two lemmas, we can prove an equation relating *fold1* to *fold*

$$\mathit{fold1}\ (\cdot)\ (\{a\} \cup A) = \mathit{fold}\ (\cdot)\ \mathit{id}\ a\ A$$

where again A must be finite and $a \notin A$. The recursion rule (3) follows easily. If $(\cdot)$ is idempotent, then $a \notin A$ can be dropped. The same holds for *fold*, but it is less useful there, because few applications of *fold* involve idempotent operators.

4.1 Properties

In the sequel, all sets are implicitly assumed to be finite and non-empty.

There are fewer general properties of *fold1* than of *fold* because *fold1* has fewer parameters. Of the *fold* lemmas in §3.4, only a single one still makes sense here: provided $A \cap B = \emptyset$, *fold1* distributes over union.

$$\mathit{fold1}\ (\cdot)\ (A \cup B) = \mathit{fold1}\ (\cdot)\ A \cdot \mathit{fold1}\ (\cdot)\ B$$

For *fold*, this distributive law is the corollary of the more general lemma (2), which does not hold for *fold1*.

If $(\cdot)$ is idempotent as well, the premise $A \cap B = \emptyset$ in the distributive law can be dropped. In fact, idempotence of $(\cdot)$ makes *fold1* come into its own.

We will now examine properties of *fold1* in various ordered structures (for details see the literature [4]). These structures are again formalized by locales in Isabelle, but our presentation will stay on an abstract mathematical level.

4.2 Semilattices

In this subsection, we assume $(\cdot)$ is not just AC but also *idempotent*: $x \cdot x = x$. This means we are in a semilattice. To obtain the order-theoretic view, we define the symbol $\sqsubseteq$ by

$$(x \sqsubseteq y) = (x \cdot y = x).$$

The semi-lattice law $(x \sqsubseteq y \cdot z) = (x \sqsubseteq y \wedge x \sqsubseteq z)$ has a nice generalization in terms of *fold1*:

$$(x \sqsubseteq \mathit{fold1}\ (\cdot)\ A) = (\forall\, a{\in}A.\ x \sqsubseteq a)$$

The dual property $(x \cdot y \sqsubseteq z) = (x \sqsubseteq z \vee y \sqsubseteq z)$ holds iff the ordering $\sqsubseteq$ is linear. Then, it can be generalized to

$$(\mathit{fold1}\ (\cdot)\ A \sqsubseteq x) = (\exists\, a{\in}A.\ a \sqsubseteq x).$$

Note that only the left-to-right direction of this equivalence requires linearity. The other direction is valid in arbitrary semilattices:

$$a \in A \Longrightarrow \mathit{fold1}\ (\cdot)\ A \sqsubseteq a, \tag{4}$$

4.3 Lattices

Frequently, the semilattice is in fact a lattice: there is not just an infimum ($(\cdot)$ above) but also a supremum, which we write $\sqcap$ and $\sqcup$. With the help of *fold1*, we can define the standard extension of $\sqcap$ and $\sqcup$ to finite sets:

$$\bigsqcap A \equiv \mathit{fold1}\ (\sqcap)\ A \qquad \bigsqcup A \equiv \mathit{fold1}\ (\sqcup)\ A$$

We inherit the semilattice laws and can derive new ones from them. For example, we obtain $\bigsqcap A \sqsubseteq \bigsqcup A$ from the instances of (4) for the two semilattices $\sqcap$ and $\sqcup$.

In case of a distributive lattice, distributivity propagates from the binary to the n-ary operations. Here are two examples:

$$x \sqcup \bigsqcap A = \bigsqcap \{x \sqcup a \mid a \in A\} \qquad \bigsqcap A \sqcup \bigsqcap B = \bigsqcap \{a \sqcup b \mid a \in A \wedge b \in B\}$$

4.4 Applications

The most direct applications of *fold1* are minimum and maximum because (as noted in §2.2) many types lack a least or greatest element, which means *fold* is inappropriate. In Isabelle, we can define *Min* and *Max* as follows:

$$\mathit{Min} \equiv \mathit{fold1}\ \mathit{min} \qquad \mathit{Max} \equiv \mathit{fold1}\ \mathit{max}$$

Here, *min* and *max* are overloaded functions available on any type of class *ord*, which means the type must define an ordering $\leq$. Hence, *Min* and *Max* have type $'a\ \mathit{set} \Rightarrow 'a$, where $'a$ is of class *ord*.

We can now inherit all of the *fold1* properties described above because *Min* and *Max* form a distributive lattice. After instantiating the corresponding locales we obain, for example, the distributive law

$$\mathit{max}\ (\mathit{Min}\ A)\ (\mathit{Min}\ B) = \mathit{Min}\ \{\mathit{max}\ a\ b \mid a \in A \wedge b \in B\}$$

Functions *Min* and *Max* and their properties are now available, for example, on all numeric types (except the complex numbers) as they are linearly ordered.

In a similar manner, we can define the greatest common divisor and the least common multiple of a set of natural numbers: $\mathit{Gcd} \equiv \mathit{fold1}\ \mathit{gcd}$ and $\mathit{Lcm} \equiv \mathit{fold1}\ \mathit{lcm}$, where *gcd* and *lcm* are the binary versions. Functions *gcd* and *lcm* also form a distributive lattice, where the ordering is divisibility. They even form a complete lattice.

Finally, we consider the longest common prefix (*lcp*) of two lists:

$$\begin{aligned} \mathit{lcp}\ []\ \mathit{xs} &= [] \\ \mathit{lcp}\ \mathit{xs}\ [] &= [] \\ \mathit{lcp}\ (x\ \#\ \mathit{xs})\ (y\ \#\ \mathit{ys}) &= (\mathsf{if}\ x = y\ \mathsf{then}\ x\ \#\ \mathit{lcp}\ \mathit{xs}\ \mathit{ys}\ \mathsf{else}\ []) \end{aligned}$$

where # is the list *Cons*. Of course, the corresponding ordering is the prefix ordering. This only yields a lower semilattice: there is no greatest element and no supremum. Thus we only define $\mathit{Lcp} \equiv \mathit{fold1}\ \mathit{lcp}$.

4.5 Alternative Definitions

As in §3.3, we can use the axiom of choice to define *fold1* by the obvious recursion on the cardinality. Again, it is simpler to define *fold1* in terms of *fold*:

$$fold1\ f\ A \equiv fold\ f\ id\ (pick\ A)\ (rest\ A)$$

The main advantage of this definition is that the recursion rule (3) can now be proved by a few case distinctions, a few properties of *pick*, and equational reasoning alone, assuming we already know

$$a \in A \Longrightarrow fold\ (\cdot)\ g\ z\ A = g\ a \cdot fold\ (\cdot)\ g\ z\ (A - \{a\})$$
$$x \cdot fold\ (\cdot)\ g\ z\ A = fold\ (\cdot)\ g\ (x \cdot z)\ A$$

both of which are natural lemmas for *fold*. This proof of (3) has the advantage over the one via *fold1Set* that it does not require any special purpose lemmas.

Neither PVS nor HOL4 provide an analogue of *fold1*. Both systems define the minimum and maximum of a set directly. Other functions like the above *Gcd* and *Lcp* would need to be defined separately.

HOL4 defines *Min* and *Max* only for sets of natural numbers. An advantage of the HOL4 approach is that *Min* also works for infinite sets. Folding does not make sense for infinite sets unless we introduce some notion of limit.

Let us compare the different approaches and assume we are interested in finite sets only. Then *fold1* has the advantage of generality over special purpose *Min* and *Max* definitions. The shortest definitions and proofs are obtained by defining *fold1* with the help of *fold* and choice. The inductive definition is a bit lengthier but not significantly so, and avoids choice. But even if one is just interested in *Min* and *Max*, their definition by description is not ideal. It is simpler by far to derive the characteristic properties of *Min* and *Max* from the recursion equations than the other way around. This can be seen by comparing the Isabelle and HOL4 proofs.

5 Conclusions

Recursive function definitions over finite sets are not difficult to justify. The mathematics is simple. We have taken pains to ensure that the machine formalization is simple too, while avoiding any dependence on the axiom of choice. The applications we have discussed are cardinality, sum and product over sets. For the case of non-empty sets, we have discussed the maximum and minimum operators.

One question we have not discussed at all is definability. It is easily seen that not every function on finite sets is definable by means of fold: if $F\ A \equiv card\ A \leq 1$ were definable as $F \equiv fold\ (\cdot)\ g\ e$ for suitable $(\cdot)$, g and e, then it would follow that $e = F\ \emptyset = True$, $g\ x \cdot e = F\ \{x\} = True$ and hence $False = F\{x,y\} = g\ x \cdot g\ y \cdot e = g\ x \cdot e = True$. On the other hand *fold* can trivially define any homomorphism from the finite sets viewed as a $(\emptyset,\{-\},\cup)$-algebra into a $(e,g,\cdot)$-algebra: $F \equiv fold\ (\cdot)\ g\ e$. But being a homomorphism this implies that $\cdot$

must satisfy all union laws, in particular idempotence: $F\ A = F(A \cup A) = F\ A \cdot F\ A$. Hence set sum and product are not homomorphisms but still defineable. This shows that the definability question is outside the scope of this paper and requires a separate study analogous to the work of Gibbons *et al.* [6] for lists.

References

1. C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *International Workshop TYPES 2003*, LNCS 3085, pages 34–50. Springer, 2004.
2. V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, editors, *Automata, Languages and Programming (ICALP91)*, volume 510 of *LNCS*, pages 60–75. Springer, 1991.
3. C.-T. Chou. Generalizing an associative and commutative operation with identity to finite sets. ftp://ftp.cl.cam.ac.uk/hvg/hol88/contrib/aci/, 1992. HOL88 formal development.
4. B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
5. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
6. J. Gibbons, G. Hutton, and T. Altenkirch. When is a Function a Fold or an Unfold? In *Proc. 4th International Workshop on Coalgebraic Methods in Computer Science*, volume 44.1 of *Electronic Notes in Theoretical Computer Science*, 2001.
7. The HOL4 Theorem Prover. http://hol.sf.net.
8. G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.
10. A. Ohori, P. Buneman, and V. Tannen. Database programming in machiavelli - a polymorphic language with static type inference. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. 1989 ACM SIGMOD Intl. Conf. Management of Data*, pages 46–57. ACM Press, 1989.
11. L. C. Paulson. Defining functions on equivalence classes. *ACM Transactions on Computational Logic*. in press.
12. L. C. Paulson. Organizing numerical theories using axiomatic type classes. *Journal of Automated Reasoning*, 2005. in press.
13. PVS Specification and Verification System. http://pvs.csl.sri.com/.

Proof Pearl: Using Combinators to Manipulate `let`-Expressions in Proof

Michael Norrish[1] and Konrad Slind[2]

[1] National ICT, Australia
Michael.Norrish@nicta.com.au
[2] School of Computing, University of Utah
slind@cs.utah.edu

Abstract. We discuss methods for dealing effectively with `let`-bindings in proofs. Our contribution is a small set of unconditional rewrite rules, found by the bracket abstraction translation from the λ-calculus to combinators. This approach copes with the usual HOL encodings of paired abstraction, ensures that bound variable names are preserved, and uses only conventional simplification technology.

1 Introduction

A characteristic feature of functional programming is the binding of values using expressions of the form `let` $v = M$ `in` N; the value of M is calculated once and may be used multiple times in N. In theorem-proving, `let`-terms are also useful, for several reasons. First, `let`-terms are obviously needed when modelling functional programs in logic. Second, sharing of sub-expressions by `let` is useful for readability during interactive proof, and can also be helpful in controlling expression size in automated proof.

In interactive proof, the most common operation applied to `let`-terms is to eliminate them. In many cases, this is quite easy to do: a `let`-term is transformed into a β-redex which is then β-reduced. This amounts to removing the sharing introduced by the `let`. However, at times one wants to eliminate a `let` but keep the sharing. This can be awkward, particularly when the `let`-term binds a tuple of variables.

We know of three ways to eliminate `let`s without losing sharing: the first asserts appropriate assumptions; the second uses a single higher order rewrite rule, supported by non-local matching; and the third, which we now prefer, uses local matching to apply a small set of rewrite rules. These rewrites have been found by applying the notion of bracket abstraction [4] from combinatory logic. For interactive proof, an important consequence of this is that user-specified bound variable names are preserved.

Example. In the classic functional programming presentation of Quicksort (see Figure 1), the function partition, defined in terms of a helper function part, is

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 397–408, 2005.

```
part P [] ℓ1 ℓ2 = (ℓ1, ℓ2)
part P (h :: rst) ℓ1 ℓ2 =
  if P h then part P rst (h :: ℓ1) ℓ2 else part P rst ℓ1 (h :: ℓ2)

partition P ℓ = part P ℓ [] []

qsort ord [] = []
qsort ord (h :: t) =
  let (ℓ1, ℓ2) = partition (λy. ord y h) t
  in qsort ord ℓ1 ++ [h] ++ qsort ord ℓ2
```

Fig. 1. Quicksort

used to divide the list around a pivot element. The paired `let`-expression in qsort binds the resulting pair of lists in preparation for the two recursive calls. The definition of qsort generates the following induction theorem:

$$\begin{array}{l}\vdash \forall P.\ (\forall ord.\ P\ ord\ [\,]) \wedge \\ \quad (\forall ord\ h\ t. \\ \quad\quad (\forall \ell_1 \ell_2.\ ((\ell_1,\ell_2) = \mathsf{partition}\ (\lambda y.\ ord\ y\ h)\ t) \supset P\ ord\ \ell_2) \wedge \\ \quad\quad (\forall \ell_1 \ell_2.\ ((\ell_1,\ell_2) = \mathsf{partition}\ (\lambda y.\ ord\ y\ h)\ t) \supset P\ ord\ \ell_1)\ \supset \\ \quad\quad P\ ord\ (h :: t)) \\ \quad \supset \forall v\ v_1.\ P\ v\ v_1\end{array} \tag{1}$$

Now suppose that the notion of one list being a permutation of another has been formalised. We wish to prove that qsort permutes its input: the goal

$$\forall ord\ \ell.\ \mathsf{perm}\ \ell\ (\mathsf{qsort}\ ord\ \ell)$$

asserts that (qsort $ord\ \ell$) is always a permutation of ℓ. The proof begins with an application of the induction theorem (1). Two cases arise: a base case which is easily dispensed with, and an induction case:[1]

$$\begin{array}{l}\mathsf{perm}\ (h :: t)\ (\mathsf{qsort}\ ord\ (h :: t)) \\ \hline 1.\ \forall \ell_1 \ell_2.\ ((\ell_1,\ell_2) = \mathsf{partition}(\lambda y.\ ord\ y\ h)\ t) \supset \mathsf{perm}\ \ell_1\ (\mathsf{qsort}\ ord\ \ell_1) \\ 2.\ \forall \ell_1 \ell_2.\ ((\ell_1,\ell_2) = \mathsf{partition}(\lambda y.\ ord\ y\ h)\ t) \supset \mathsf{perm}\ \ell_2\ (\mathsf{qsort}\ ord\ \ell_2)\end{array}$$

There are two induction hypotheses, mirroring the recursive calls of qsort. If we now expand the definition of qsort in the goal, we obtain the new goal

$$\begin{array}{l}\mathsf{perm}\ (h :: t)\ (\texttt{let}\ (\ell_1,\ell_2) = \mathsf{partition}\ (\lambda y.\ ord\ y\ h)\ t \\ \qquad\qquad \texttt{in}\ \ \mathsf{qsort}\ ord\ \ell_1\ \ ++\ [h]\ \ ++\ \mathsf{qsort}\ ord\ \ell_2) \\ \hline 1.\ \forall \ell_1 \ell_2.\ ((\ell_1,\ell_2) = \mathsf{partition}(\lambda y.\ ord\ y\ h)\ t) \supset \mathsf{perm}\ \ell_1\ (\mathsf{qsort}\ ord\ \ell_1) \\ 2.\ \forall \ell_1 \ell_2.\ ((\ell_1,\ell_2) = \mathsf{partition}(\lambda y.\ ord\ y\ h)\ t) \supset \mathsf{perm}\ \ell_2\ (\mathsf{qsort}\ ord\ \ell_2)\end{array}$$

[1] The goal appears above the line and (numbered) assumptions below the line.

We can now make our point. If it was somehow possible to lift the subformula $(\ell_1, \ell_2) = \mathsf{partition}\ (\lambda y.\ ord\ y\ h)\ t$ from within the goal and place it on the assumptions, we could move to the goal

$$\frac{\mathsf{perm}\ (h :: t)\ (\mathsf{qsort}\ ord\ \ell_1\ +\!\!+\ [h]\ +\!\!+\ \mathsf{qsort}\ ord\ \ell_2)}{\begin{array}{l}1.\ \forall \ell_1 \ell_2.\ ((\ell_1, \ell_2) = \mathsf{partition}(\lambda y.\ ord\ y\ h)\ t) \supset \mathsf{perm}\ \ell_1\ (\mathsf{qsort}\ ord\ \ell_1)\\ 2.\ \forall \ell_1 \ell_2.\ ((\ell_1, \ell_2) = \mathsf{partition}(\lambda y.\ ord\ y\ h)\ t) \supset \mathsf{perm}\ \ell_2\ (\mathsf{qsort}\ ord\ \ell_2)\\ 3.\ (\ell_1, \ell_2) = \mathsf{partition}\ (\lambda y.\ ord\ y\ h)\ t\end{array}}$$

and use the inductive hypotheses to obtain

$$\frac{\mathsf{perm}\ (h :: t)\ (\mathsf{qsort}\ ord\ \ell_1\ +\!\!+\ [h]\ +\!\!+\ \mathsf{qsort}\ ord\ \ell_2)}{\begin{array}{l}1.\ \forall \ell_1 \ell_2.\ ((\ell_1, \ell_2) = \mathsf{partition}(\lambda y.\ ord\ y\ h)\ t) \supset \mathsf{perm}\ \ell_1\ (\mathsf{qsort}\ ord\ \ell_1)\\ 2.\ \forall \ell_1 \ell_2.\ ((\ell_1, \ell_2) = \mathsf{partition}(\lambda y.\ ord\ y\ h)\ t) \supset \mathsf{perm}\ \ell_2\ (\mathsf{qsort}\ ord\ \ell_2)\\ 3.\ (\ell_1, \ell_2) = \mathsf{partition}\ (\lambda y.\ ord\ y\ h)\ t\\ 4.\ \mathsf{perm}\ \ell_1\ (\mathsf{qsort}\ ord\ \ell_1)\\ 5.\ \mathsf{perm}\ \ell_2\ (\mathsf{qsort}\ ord\ \ell_2)\end{array}}$$

From this situation, only some simple theorems about permutations are needed to finish the proof.

However, it is not clear how this step of lifting the `let`-binding out of the goal can be accomplished. For example, naïvely rewriting with the definition of `let` (for which, see below) is disastrous: at best, it results in duplication of the occurrence of $\mathsf{partition}\ (\lambda y.\ ord\ y\ h)\ t$ in the goal, which is unpleasant to read and makes it difficult to apply the induction hypotheses. Instead, we want to eliminate the `let` while moving the binding for ℓ_1 and ℓ_2 to the assumptions. In order to understand the issues, we now address how `let`-terms are represented in HOL.

1.1 Representing Variable-Binding Operators

Semantically, a `let`-term is just a suspended function application.

$$\vdash \mathsf{LET}\ f\ x = f\ x$$

The real work occurs in the HOL parser, which translates the surface syntax of `let`-terms into applications of the LET constant. For example, `let` $v = M$ `in` N is mapped to $\mathsf{LET}\ (\lambda v.\ N)\ M$. This translation is inverted by the prettyprinter.

The representation of paired λ-abstractions is more complex: an abstraction of the form $(\lambda(x, y).\ \ldots x \ldots y)$ is a convenient notation for a function with a product type as its domain. Such abstractions are redundant because the accessor functions FST and SND could be used instead: the abstraction above is equivalent to $(\lambda p.\ \ldots (\mathsf{FST}\ p) \ldots (\mathsf{SND}\ p))$. However, the pattern-matching style is far more readable and is supported in HOL by an encoding. The paired abstraction syntax is mapped to an application of the UNCURRY combinator, defined as follows:

$$\vdash \mathsf{UNCURRY}\ f\ p = f\ (\mathsf{FST}\ p)\ (\mathsf{SND}\ p)\ .$$

If f has the form $(\lambda x\, y.\, M)$, then HOL prints UNCURRY f as $(\lambda(x, y).M)$. UNCURRY terms can be nested, producing abstractions over three or more variables. For example

$$\begin{aligned}\mathsf{UNCURRY}\ (\lambda x.\ \mathsf{UNCURRY}\ (\lambda y\, z.\ M)) &= (\lambda(x, (y, z)).\ M)\\ \mathsf{UNCURRY}\ (\mathsf{UNCURRY}\ (\lambda x\, y\, z.\ M)) &= (\lambda((x, y), z).\ M)\end{aligned}$$

If the first argument to LET is a paired abstraction, then the `let`-term is printed so as to suggest the binding of multiple values at once. For example, LET $(\lambda(x, y).\ N)\ M$ is printed as `let` $(x, y) = M$ `in` N.

2 Eliminating `let`-Terms

In general, we wish to move from

$$\frac{P\ (\texttt{let}\ (x_1, \ldots, x_n) = M\ \texttt{in}\ N)}{\cdots \text{assumptions} \cdots} \qquad \text{to} \qquad \frac{P\ N}{\begin{array}{c}\cdots \text{assumptions} \cdots\\ (x_1, \ldots, x_n) = M\end{array}}$$

where N presumably includes at least some of the variables $x_1 \ldots x_n$, and where none of the x_i or the variables in M are bound by the enclosing context P. We now discuss three alternative ways to achieve this.

2.1 Approach 1: Making Assumptions Directly

As long as variable x is fresh for the goal, it is valid to extend a goal's assumptions with a fresh assumption of the form $x = e$, for an arbitrary e that does not mention x. Similarly, if e has a product type, one can add an assumption of the form $(x_1, \ldots, x_n) = e$. These steps are justified by the theorems

$$\vdash P = (\forall x.\ (x = e) \supset P)$$

(with $x \notin (e, P)$) and

$$\vdash (\forall p.\ P\ (p)) = (\forall a\, b.\ P\ (a, b)) \tag{2}$$

which turns universal quantification over variables of product type into multiple universal quantifications. When the goal looks like

$$\frac{P\ (\texttt{let}\ (x_1, \ldots, x_n) = M\ \texttt{in}\ N)}{\begin{array}{l}\cdots \text{assumptions} \cdots\\ (x_1, \ldots, x_n) = M\end{array}}$$

the free occurrence of M in the goal can be replaced by the tuple $(x_1, \ldots, x_n)$, the `let`-expression removed by rewriting with the definition of LET, and a β-reduction performed. This then produces the desired goal. (Performing a β-reduction for paired abstractions applied to tuples of the correct arity is simply

a matter of rewriting with the definition of UNCURRY and then performing normal β-reductions.)

This approach is conceptually simple, and would be relatively easy to implement. The only complexity would be the requirement to check the various freeness conditions mentioned above. This analysis is non-local: the movement of the `let`-expressions depends on their surrounding context. Another limiting factor is that the step of explicitly making an assumption can make it awkward to integrate this approach with a term rewriter.

2.2 Approach 2: Lifting `let` with One Higher Order Rewrite

Another approach to lifting `let`-expressions makes the theorems used in the previous approach more explicit. We might use the following theorem:

$$\vdash P\ (\texttt{let}\ (x, y) = M\ \texttt{in}\ N\ x\ y) = (\forall x\ y.\ ((x, y) = M) \supset P\ (N\ x\ y)) \quad (3)$$

This theorem lifts the nested `let`-expression to the top-level of the goal, where the `let` is replaced by an equational binding on the variables x and y. In the Quicksort example, the instantiation for P would be perm $(h :: t)$.

Further, it is easy enough to generalise the theorem above to one that works for arbitrary n-tuples:

$$\vdash P\ (\mathsf{LET}\ f\ M) = \forall v.\ (v = M) \supset P\ (f\ v) \quad (4)$$

This is not quite enough because we do not have a tuple of variables equated with M, but rather the single variable v. If the f of the above theorem is not a paired abstraction, then v will not have a product type, and it is trivial to move the equality into the assumptions. Otherwise f is a paired abstraction, so we must first rewrite with theorem (2) in order to introduce variables for each element of the product type.

While rewrite rules (3) and (4) are elegant and easy to prove, they are not always easy to apply. In the Quicksort example, rewriting with either works well, but only accidentally: P can be found with a simple higher-order match. However, had the `let`-term been the first argument to perm, instantiating the rewrite rule would have moved beyond what is possible with usual pattern-based higher-order matching [5]. In similar situations to this, the Isabelle system [6] uses its "splitting" technology (also used for case-splits over types) rather than normal rewriting. There for example, the splitting rule for UNCURRY is

$$\vdash P\ (\mathsf{UNCURRY}\ f\ p) = \forall x\ y.\ ((x, y) = p) \supset P\ (f\ x\ y)$$

where P must be found by means other than a simple match. Note that this rule preserves sharing if the argument to a paired abstraction is not a tuple of matching arity, but does not otherwise help in the preservation of `let`-induced sharing.

Just as the 'Making Assumptions Directly' approach has to perform global analyses of the goal in order to make the correct assumptions, so too must this

one-step higher-order rewrite approach, even if this analysis is embodied in an Isabelle-style splitter. One other problem with this approach is that it may be difficult to ensure the preservation of bound variable names. Certainly, Isabelle's splitter for UNCURRY does not currently manage this (bound names in the abstraction f are replaced by x and y).

Our preferred approach, described next, is a system that preserves bound variable names, and works entirely locally, requiring only higher-order pattern matching and the application of simple rewrite rules.

2.3 Approach 3: Lifting `let` by Multiple Combinator Rewrites

Superficially, the general problem of moving `let`-expressions upwards is solved by the theorems:

$$\vdash f\ (\mathsf{LET}\ g\ M) = \mathsf{LET}\ (\lambda x.\ f\ (g\ x))\ M \tag{5}$$

$$\vdash (\mathsf{LET}\ f\ M)\ N = \mathsf{LET}\ (\lambda x.\ f\ x\ N)\ M \tag{6}$$

Erasing the LETs, and performing β-reduction makes it clear that the equations are semantically valid. They also satisfy the important syntactic criterion that the LET-expressions' second arguments (M in both cases) remain as second arguments to LETs on the right-hand sides. In this way, the "pending computations" represented by the M's remain pending, that is, not substituted through the body of the LETs' first arguments.

The following demonstrates the application of (5) as a first-order rewrite:

$$\begin{array}{ll} & \neg(\texttt{let}\ v = M\ \texttt{in}\ (v \wedge p \vee r)) \\ = & \neg(\mathsf{LET}(\lambda v.\ v \wedge p \vee r)\ M) \\ = & \mathsf{LET}\ (\lambda x.\ \neg((\lambda v.\ v \wedge p \vee r)\ x))\ M \qquad \text{by (5)} \\ = & \texttt{let}\ x = M\ \texttt{in}\ \neg((\lambda v.\ v \wedge p \vee r)\ x) \\ =_\beta & \texttt{let}\ x = M\ \texttt{in}\ \neg(x \wedge p \vee r) \end{array}$$

While this has moved the LET to the top of the expression, and has kept M in the right position relative to it, the rewriting has obliterated the original choice of bound names. In this case at least, there is a fairly straightforward fix: the rewriting engine could be modified so that when it generates fresh bound names (x in this case), and they appear as arguments to λ-abstractions, it changes the new bound name to be the same as the bound name of the abstraction. In this way, the abstraction $(\lambda v.\ v \wedge p \vee r)$ above will force the new bound name to become v. After this α-conversion, the term can safely be β-reduced.

Unfortunately, this technology does not work when the abstraction under the LET is paired. For example:

$$\begin{array}{ll} & \neg(\texttt{let}\ (u, v) = M\ \texttt{in}\ u \wedge v \vee r) \\ = & \neg(\mathsf{LET}(\mathsf{UNCURRY}\ (\lambda u\ v.\ u \wedge v \vee r))\ M) \\ = & \mathsf{LET}\ (\lambda x.\ \neg(\mathsf{UNCURRY}\ (\lambda u\ v.\ u \wedge v \vee r)x))\ M \qquad \text{by (5)} \\ = & \texttt{let}\ x = M\ \texttt{in}\ (\lambda(u, v).\ u \wedge v \vee r)\ x \end{array}$$

Although this term can be further modified by deductive steps to obtain $\texttt{let}\ (u, v) = M\ \texttt{in}\ u \wedge v \vee r$, that involves some detailed programming. Instead, our approach is to arm the standard system simplifier with a sufficient set of rewrite rules. One simple principle guides the development:

> *Never use rewrite rules that introduce fresh bound names on their right-hand sides.*

The key to enforcing this is to use rewrite rules phrased in terms of the combinators B and C. For example, the new versions of the rules for moving LETs upwards are

$$\vdash f\ (\mathsf{LET}\ g\ M) = \mathsf{LET}\ (\mathsf{B}\ f\ g)\ M \tag{7}$$

$$\vdash (\mathsf{LET}\ f\ M)\ N = \mathsf{LET}\ (\mathsf{C}\ f\ N)\ M \tag{8}$$

where[2]

$$\vdash \mathsf{B}\ f\ g\ x = f\ (g\ x)$$
$$\vdash \mathsf{C}\ f\ x\ y = f\ y\ x$$

Also needed are rules to allow abstractions to move up over the combinators.

$$\vdash \mathsf{B}\ f\ (\lambda x.\ g\ x) = (\lambda x.\ f\ (g\ x)) \tag{9}$$

$$\vdash \mathsf{C}\ (\lambda x.\ f\ x)\ y = (\lambda x.\ f\ x\ y) \tag{10}$$

Without these, rewriting our first example above would stop at[3]

$$\mathsf{LET}\,(\mathsf{B}\ (\neg)\ (\lambda v.\ v \wedge p \vee r))\ M$$

These new rules respect the principle of not introducing new bound names. Note that application of these rules requires higher-order rewriting [5]; moreover, rewriting must also take care to preserve the bound names it encounters when matching the left-hand side.

The development so far has dealt with the problem of lifting LETs with normal abstractions underneath them.

2.4 Dealing with Tupled `let`-Bindings

To handle LETs with paired abstractions, we continue to use (7) and (8). Rules for B and C that behave properly when applied to functions formed by use of UNCURRY are also required. First the rule for B:

$$\vdash \mathsf{B}\ f\ (\mathsf{UNCURRY}\ g) = \mathsf{UNCURRY}\ (\mathsf{B}\ (\mathsf{B}\ f)\ g) \tag{11}$$

[2] The combinator B is often written as an infix $\circ$.

[3] Note that $\neg$ appears here as an unapplied function, represented by enclosing it in extra parentheses.

In the example that went wrong above, our new technology does the following:

$$\begin{aligned}
&\neg(\texttt{let}\ (u,v) = M\ \texttt{in}\ u \wedge v \vee r)\\
&= \neg(\mathsf{LET}\ (\mathsf{UNCURRY}(\lambda u\ v.\ u \wedge v \vee r))\ M)\\
&= \mathsf{LET}\ (\mathsf{B}\ (\neg)\ (\mathsf{UNCURRY}(\lambda u\ v.\ u \wedge v \vee r)))\ M && \text{by (7)}\\
&= \mathsf{LET}\ (\mathsf{UNCURRY}\ (\mathsf{B}\ (\mathsf{B}\ (\neg))\ (\lambda u\ v.\ u \wedge v \vee r)))\ M && \text{by (11)}\\
&= \mathsf{LET}\ (\mathsf{UNCURRY}\ (\lambda u.\ \mathsf{B}\ (\neg)\ (\lambda v.\ u \wedge v \vee r)))\ M && \text{by (9)}\\
&= \mathsf{LET}\ (\mathsf{UNCURRY}\ (\lambda u\ v.\ \neg(u \wedge v \vee r)))\ M && \text{by (9)}\\
&= \texttt{let}\ (u,v) = M\ \texttt{in}\ \neg(u \wedge v \vee r)
\end{aligned}$$

The beauty of this approach is that precisely the right number of B combinators will be generated underneath the UNCURRY term to "consume" the unitary bound variables of the curried abstractions underneath. For example, note that if g in (11) is itself another UNCURRY term, then the argument to UNCURRY in the right-hand side of (11) will itself be a match for the same rewrite, and another B will be generated.

The rule for C is

$$\vdash \mathsf{C}\ (\mathsf{UNCURRY}\ f)\ x = \mathsf{UNCURRY}\ (\mathsf{C}\ (\mathsf{B}\ \mathsf{C}\ f)\ x) \tag{12}$$

The question remains: how were (11) and (12) found?

We demonstrate the derivation of (12). This equation relates functions. That means the process quite reasonably begins with an η-expansion. Then, because the functions are over pairs, we can immediately introduce a paired abstraction (an UNCURRY term) at the top level:

$$\begin{aligned}
&\mathsf{C}\ (\mathsf{UNCURRY}\ f)\ x\\
&= \lambda p.\ \mathsf{C}\ (\mathsf{UNCURRY}\ f)\ x\ p\\
&= \lambda(u,v).\ \mathsf{C}\ (\mathsf{UNCURRY}\ f)\ x\ (u,v)
\end{aligned}$$

The definitions of C and UNCURRY then simplify this to $(\lambda(u,v).\ f\ u\ v\ x)$, which is really $\mathsf{UNCURRY}\ (\lambda u\, v.\ f\ u\ v\ x)$. At this point, bracket abstraction (see Appendix A) is used to remove the bindings over u and v:

$$\begin{aligned}
&\mathsf{UNCURRY}\ (\lambda u\, v.\ f\ u\ v\ x)\\
&= \mathsf{UNCURRY}\ (\lambda u.\ [v](f\ u\ v\ x))\\
&= \mathsf{UNCURRY}\ (\lambda u.\ \mathsf{C}\ (f\ u)\ x)\\
&= \mathsf{UNCURRY}\ ([u]((f\ u)\ x))\\
&= \mathsf{UNCURRY}\ (\mathsf{C}\ (\mathsf{B}\ \mathsf{C}\ f)\ x)
\end{aligned}$$

The derivation of (11) is similar.

We now have equations for pushing B and C under both normal abstractions (9, 10), and paired abstractions (11, 12). In conjunction with the rules for LET movement (7, 8), this supports a smooth rewriting strategy for moving any sort of `let`-expression to the outermost position of a term.

2.5 Eliminating `let`

The higher-order rewrites of our second approach did more than simply move `let`-expressions upwards. They also turned the `let`-binding into a universally

quantified implication. From there, a standard HOL tactic could be used to push the antecedent of the implication (the `let`-bindings) into the assumptions of the goal. To calculate the combinator version, we start with a naïve rewrite to accomplish this on a term of type bool:

$$\vdash \mathsf{LET}\ f\ M = \forall x.\ (x = M) \supset f(x)$$

When performing bracket abstraction on this term, we need the S combinator:

$$\vdash \mathsf{S}\ f\ g\ x = (f\ x)\ (g\ x)$$

Bracket abstraction then turns our naïve rewrite into

$$\vdash \mathsf{LET}\ f\ M = (\forall)(\mathsf{S}\ (\mathsf{B}\ (\supset)\ (\mathsf{C}\ (=)\ M))\ f)$$

(Combinators truly are the machine-code of functional programming! Again, we use extra parentheses around symbols ($\forall$, $\supset$ and $=$ here) to make it clear that these are not being used in their binding or infix forms.)

Now that S has made its appearance, we need a rule to move abstractions past it. The rule for normal abstractions is obvious

$$\vdash \mathsf{S}\ f\ (\lambda x.\ g\ x) = \lambda x.\ (f\ x)\ (g\ x) \tag{13}$$

For paired abstractions, the rule is more combinatory machine code:

$$\vdash \mathsf{S}\ f\ (\mathsf{UNCURRY}\ g) = \mathsf{UNCURRY}\ (\mathsf{S}\ (\mathsf{B}\ \mathsf{S}\ (\mathsf{B}\ (\mathsf{B}\ f)\ (,)))\ g)$$

where $(,)$ is the (curried) operator that produces pairs. Its presence is required because the function f expects to be applied to a pair.

Finally, we also require a rule to deal with a universal quantifier being applied to a paired abstraction:

$$\vdash (\forall)\ (\mathsf{UNCURRY}\ f) = (\forall)\ (\mathsf{B}\ (\forall)\ f)$$

2.6 Applying `let`-Bindings in Proof

Our basic premise is that `let`-bound definitions should be preserved, rather than substituted out; thus, `let`-bindings are lifted out of the goal and placed on the assumptions. Whenever the `let`-expression is over a normal, unitary abstraction, the new assumption is of the form $v = M$, where v is a variable. Such assumptions are grist to the simplifier's mill: it uses them as additional rewrite rules and eliminates all of the goal's occurrences of v. But this is a source of frustration: the simplifier has undone all of the work performed in getting the `let`-bound definitions into the assumptions in the first place.

One approach to this situation is to store the equation in the assumptions as $M = v$. This has the advantage that simplification will not only leave occurrences of v alone, but also possibly find fresh occurrences of M in the goal, and replace them with v. Unfortunately, this strategy is not good enough to to leave the

bindings undisturbed by some aggressive simplification tactics: these methods look for assumed equalities oriented in either direction, and eliminate them. To remedy this, equalities arising from `let`-expressions are *tagged*, so that they appear in the assumptions as $\mathsf{Abbrev}(v = M)$. Semantically, Abbrev is the identity function on booleans. Its role here is simply to maintain equalities that might otherwise be eliminated. If a user wishes to eliminate abbreviations, there are tactics for doing so on an abbreviation-by-abbreviation basis, or all at once. Automated reasoners eliminate the abbreviations as part of their pre-processing.

It is easy to accommodate Abbrev in our system: we modify the rule for the final elimination of LET to

$$\vdash \mathsf{LET}\ f\ M = (\forall)(\mathsf{S}\ (\mathsf{B}\ (\supset)\ (\mathsf{B}\ \mathsf{Abbrev}\ (\mathsf{C}\ (=)\ M)))\ f)$$

This yields the final collection of rules, found in Figure 2.

$$\begin{aligned}
\vdash f\ (\mathsf{LET}\ g\ M) &= \mathsf{LET}\ (\mathsf{B}\ f\ g)\ M\\
\vdash (\mathsf{LET}\ f\ M)\ y &= \mathsf{LET}\ (\mathsf{C}\ f\ y)\ M\\
\vdash \mathsf{LET}\ f\ M &= (\forall)(\mathsf{S}\ (\mathsf{B}\ (\supset)\ (\mathsf{B}\ \mathsf{Abbrev}\ (\mathsf{C}\ (=)\ M)))\ f)\\
\vdash \mathsf{B}\ f\ g\ x &= f\ (g\ x)\\
\vdash \mathsf{C}\ f\ x\ y &= f\ y\ x\\
\vdash \mathsf{S}\ f\ g\ x &= (f\ x)\ (g\ x)\\
\vdash \mathsf{B}\ f\ (\lambda x.\ g\ x) &= (\lambda x.\ f\ (g\ x))\\
\vdash \mathsf{C}\ (\lambda x.\ f\ x)\ y &= (\lambda x.\ f\ x\ y)\\
\vdash \mathsf{S}\ f\ (\lambda x.\ g\ x) &= (\lambda x.\ (f\ x)\ (g\ x))\\
\vdash \mathsf{B}\ f\ (\mathsf{UNCURRY}\ g) &= \mathsf{UNCURRY}\ (\mathsf{B}\ (\mathsf{B}\ f)\ g)\\
\vdash \mathsf{C}\ (\mathsf{UNCURRY}\ f)\ x &= \mathsf{UNCURRY}\ (\mathsf{C}\ (\mathsf{B}\ \mathsf{C}\ f)\ x)\\
\vdash \mathsf{S}\ f\ (\mathsf{UNCURRY}\ g) &= \mathsf{UNCURRY}\ (\mathsf{S}\ (\mathsf{B}\ \mathsf{S}\ (\mathsf{B}\ (\mathsf{B}\ f)\ (,)))\ g)\\
\vdash (\forall)\ (\mathsf{UNCURRY}\ f) &= (\forall)\ (\mathsf{B}\ (\forall)\ f)
\end{aligned}$$

Fig. 2. All rewrites needed to lift `let`-expressions up and out of HOL goals. Those rewrites involving abstractions (λ) must use higher-order matching. The others may be first-order.

2.7 Limitations

We have not presented a rule in the system to lift a `let`-term over a λ-abstraction. With an expression such as

$$\lambda x.\ \texttt{let}\ y = 3\ \texttt{in}\ x + y$$

it would be nice to have the system rewrite this to

$$\texttt{let}\ y = 3\ \texttt{in}\ (\lambda x.\ x + y)$$

This is possible because the `let`-term's 'pending' computation has no dependence on the abstraction's bound variable. A suitable theorem expressing this transformation is

$$\vdash (\lambda x.\ \mathsf{LET}\ (f\ x)\ M) = \mathsf{LET}\ (\mathsf{C}\ f)\ M$$

(This rule needs to be accompanied by rules for dealing with the application of C to various forms of abstraction. Deriving these is left as an exercise for the reader.) Unfortunately, including this theorem as a rewrite rule immediately leads to non-termination. The distinct normal forms

$$\begin{array}{l}(\texttt{let}\ y = N\ \texttt{in let}\ x = M\ \texttt{in}\ P\ x\ (Q\ y)) \qquad \text{and}\\ (\texttt{let}\ x = M\ \texttt{in let}\ y = N\ \texttt{in}\ P\ x\ (Q\ y))\end{array}$$

become inter-convertible, because they include instances of the new rewrite: the underlying representation of the first normal form above is

$$\mathsf{LET}\ (\lambda y.\ \mathsf{LET}\ (\lambda x.\ P\ x\ (Q\ y))\ M)\ N$$

Nor would the possible looping be easy to prevent: the shift from one normal form to another involves the application of a number of different rewrites. In particular, the loop could not be prevented by the simplifier's simple-minded detection of loops that occur in single rules.

3 Conclusions and Future Work

We have discussed the development of a specialised set of rewrite rules aimed at lifting arbitrarily deeply-buried `let`-bindings to the top level of a formula. Our approach allows the preservation of names, which is very important for interactive use, and has the feature that specialised deductive apparatus is not needed; for example, the rules are automatically used by the standard HOL-4 simplifier.

One might ask whether or not the system of rewrites presented here is confluent. It is not: the term $((\texttt{let}\ x = u\ \texttt{in}\ P\ x)\ (\texttt{let}\ y = v\ \texttt{in}\ Q\ y))$ can reduce to either

$$\begin{array}{l}\texttt{let}\ x = u\ \texttt{in let}\ y = v\ \texttt{in}\ P\ x\ (Q\ y) \qquad \text{or}\\ \texttt{let}\ y = v\ \texttt{in let}\ x = u\ \texttt{in}\ P\ x\ (Q\ y)\end{array}$$

both of which are normal forms.

A remaining challenge is to prove that the system of rewrites presented in Figure 2 terminates on all inputs.

References

1. Sabine Broda and Luis Damas. Compact bracket abstraction in combinatory logic. *Journal of Symbolic Logic*, 62(3):729–740, 1997.
2. Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, 1958. Two sections by William Craig.

3. Antoni Diller. Efficient mult-variate abstraction using an array representation for combinators. *Information Processing Letters*, 84(6):311–317, 2002.
4. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.
5. Tobias Nipkow. Functional unification of higher order patterns. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74, Montreal, Canada, June 1993. IEEE Computer Society Press.
6. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
7. D.A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):257–270, 1979.

A Bracket Abstraction

Bracket abstraction is a method for translating λ-calculus terms to combinator form. The basic bracket abstraction algorithm is the following, where $[x]M$ represents the abstraction of variable x from term M.

$$\begin{aligned} [x]x &= \mathsf{I} \\ [x]y &= \mathsf{K}\ y \qquad (y \text{ a constant or a variable not equal to } x) \\ [x](M\ N) &= \mathsf{S}\ ([x]M)\ ([x]N) \end{aligned}$$

If $[x]M = N$, then N has no occurrences of x and $N(x) = M$. Curry [2] adds some optimizations, dealing with the situation when the variable to be abstracted does not occur in one or more sub-terms. The following rewrites can be applied whenever a term of the form $(\mathsf{S}\ M\ N)$ is created. When more than one rule applies, the one given earlier takes precedence.

$$\begin{aligned} \mathsf{S}\ (\mathsf{K}\ M)\ (\mathsf{K}\ N) &= \mathsf{K}\ (M\ N) \\ \mathsf{S}\ (\mathsf{K}\ M)\ \mathsf{I} &= M \\ \mathsf{S}\ (\mathsf{K}\ M)\ N &= \mathsf{B}\ M\ N \\ \mathsf{S}\ M\ (\mathsf{K}\ N) &= \mathsf{C}\ M\ N \end{aligned}$$

More efficient translations [7,1,3] are known, but aren't needed for our application.

Author Index