Lecture Notes in Computer Science 5083

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

Olaf Chitil Zoltán Horváth Viktória Zsók (Eds.)

Implementation and Application of Functional Languages

19th International Symposium, IFL 2007
Freiburg, Germany, September 2007
Revised Selected Papers

Springer

Volume Editors

Olaf Chitil
Canterbury, Kent CT2 7NF, United Kingdom
E-mail: O.Chitil@kent.ac.uk

Zoltán Horváth
Viktória Zsók
Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers
1117 Budapest, Hungary
E-mail: {hz, zsv}@inf.elte.hu

Library of Congress Control Number: 2008932855

CR Subject Classification (1998): D.3, D.1.1, D.1, F.3, C.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-85372-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-85372-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12447244 06/3180 5 4 3 2 1 0

Preface

This volume contains the revised selected papers of the 19th International Symposium on Implementation and Application of Functional Languages (IFL 2007) held during September 27–29, 2007 in Freiburg, Germany.

IFL brings together researchers active in the area of functional programming, with an emphasis on the implementation and application of the same. IFL provides an annual open forum for researchers who wish to present and discuss new ideas and concepts, work in progress, preliminary results, etc. IFL covers a wide range of topics from theoretical aspects over language design and implementation towards applications and tool support.

IFL has been held throughout Europe in The Netherlands, UK, Germany, Sweden, Spain, Ireland and Hungary. In 2007—for the first time—IFL was colocated with the ACM SIGPLAN International Conference on Functional Programming (ICFP 2007) and seven affiliated workshops. IFL 2007 had a record number of 87 participants from 4 continents giving 44 presentations. These presentations were organized into 11 sessions on applications, types, contracts, compilation, parallelism, algorithms and data structures, program development, and foundations. The draft proceedings distributed during the symposium contain 44 papers and abstracts. They were published as Technical Report No. 12-07 of the Computing Laboratory, University of Kent, UK.

This volume follows the IFL tradition since 1996 in publishing a high-quality subset of papers presented at the symposium in the Springer *Lecture Notes in Computer Science* series. All participants who gave presentations at the symposium were invited to resubmit revised versions of their contributions after the symposium. We received 33 papers, each of which was reviewed by 4 members of the Programme Committee according to normal conference standards. Following an intensive one-week discussion the Programme Committee selected 15 papers for this volume.

Since 2002 the Peter Landin Prize has been awarded annually to the authors of the best paper. For 2007 the Programme Committee was pleased to award this prestigious prize to Neil Mitchell and Colin Runciman for their paper "A Supercompiler for Core Haskell".

IFL 2007 was generously sponsored by the Deutsche Forschungsgemeinschaft (DFG). The local organizers of the Programming Languages Group of the Department of Computer Science of the University of Freiburg ensured that the whole event ran smoothly. The Programme Committee members wrote 132 reviews in a short time frame. The conference management system EasyChair substantially simplified the work of the Programme Chair and communication

within the Programme Committee. Last but not least we thank all participants of IFL 2007 who made it such a successful event.

April 2008

Olaf Chitil
Zoltán Horváth
Viktória Zsók

Organization

Programme Committee

Peter Achten	Radboud University Nijmegen, The Netherlands
Kenichi Asai	Ochanomizu University, Japan
Manuel Chakravarty	University of New South Wales, Australia
Olaf Chitil (chair)	University of Kent, UK
Martin Erwig	Oregon State University, Oregon, USA
Marc Feeley	Université de Montréal, Canada
Martin Gasbichler	Zühlke Engineering AG, Switzerland
Kevin Hammond	University of St. Andrews, UK
Zoltán Horváth	Eötvös Loránd University, Budapest, Hungary
John Hughes	Chalmers University of Technology, Sweden
Ken Friis Larsen	University of Copenhagen, Denmark
Rita Loogen	Philipps-Universität Marburg, Germany
Michel Mauny	ENSTA, France
Sven-Bodo Scholz	University of Hertfordshire, UK
Clara María Segura Díaz	Universidad Complutense de Madrid, Spain
Tim Sheard	Portland State University, Oregon, USA
Glenn Strong	Trinity College, Dublin, Ireland
Doaitse Swierstra	Utrecht University, The Netherlands
Malcolm Wallace	University of York, UK

Local Organization

Markus Degen	Albert-Ludwigs-Universität Freiburg, Germany
Peter Thiemann	Albert-Ludwigs-Universität Freiburg, Germany
Stefan Wehr	Albert-Ludwigs-Universität Freiburg, Germany

Sponsoring Institutions

Deutsche Forschungsgemeinschaft (DFG)

Albert-Ludwigs-Universität Freiburg

Additional Reviewers

Tim Bauer	Emmanuel Chailloux
Jost Berthold	Chris Chambers
Edwin Brady	Mischa Dieterle

Gabriel Ditu
Péter Diviánszky
Clemens Grelck
Stephan Herhut
Joseph Hermens
Christoph Herrmann
Stefan Holdermans
Arthur Hughes
Tom Hvitved
Jan Martin Jansen
Steffen Jost
Gabriele Keller
Eric Knauel
Alexander Konovalov
Pieter Koopman
Tamás Kozsik
Roman Leshchinskiy
Nathan Linger
László Lövei
Andres Löh
Bruno Monsuez
Hidehiko Masuhara
Morten Ib Nielsen
Michael Nissen
Yolanda Ortega-Mallén
Miguel Palomino
Ricardo Peña-Marí
Yann Régis-Gianas
Didier Rémy
Fernando Rubio
Jakob Grue Simonsen
Michael Sperber
Máté Tejfel
Eric Walkingshaw
Viktória Zsók
Marko van Eekelen
John van Groningen

Table of Contents

Graph Parser Combinators

Steffen Mazanek and Mark Minas

Universität der Bundeswehr, München, Germany
{steffen.mazanek,mark.minas}@unibw.de

Abstract. A graph language can be described by a graph grammar in a manner similar to a string grammar known from the theory of formal languages. Unfortunately, graph parsing is known to be computationally expensive in general. There are quite simple graph languages that crush most general-purpose graph parsers.

In this paper we present graph parser combinators, a new approach to graph parsing inspired by the well-known string parser combinators. The basic idea is to define primitive graph parsers for elementary graph components and a set of combinators for the construction of more advanced graph parsers. Using graph parser combinators special-purpose graph parsers can be composed conveniently. Thereby, language-specific performance optimizations can be incorporated in a flexible manner.

Keywords: functional programming, graph parsing, parser combinators, visual languages.

1 Introduction

Graphs are a central data structure in computer science. Among other things, they are heavily used for modeling and specifying. For instance, we have specified visual languages using graph grammars [1]. Such graph grammars are used to define a particular graph language in analogy to string grammars known from formal language theory.

As in the setting of string grammars we are interested in solving the membership problem (checking whether a given graph belongs to a particular graph language) and parsing (finding a corresponding derivation), respectively. However, while string parsing of context-free languages can be performed in $O(n^3)$, e.g., by using the well-known algorithm of Cocke, Kasami and Younger [2], graph parsing is computationally expensive. There are even context-free graph languages the parsing of which is NP-complete [3]. Thus, a general-purpose graph parser cannot be expected to run in polynomial time for arbitrary grammars. The situation can be improved by imposing particular restrictions on the graph languages or grammars. Anyhow, even if a language can be parsed in polynomial time by a general-purpose parser, a special-purpose parser tailored to the language is likely to outperform it.

For this reason we propose a different approach to graph parsing: Graph Parser Combinators. We mainly have been inspired by the work of Hutton and Meijer

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 1–18, 2008.

[4] who have proposed monadic parser combinators for string parsing. The idea of parser combinators is much older, though. The basic principle of a parser combinator library is that primitive parsers are provided that can be combined into more advanced parsers using a set of powerful combinators.

The most prominent combinators are the sequence and the choice combinator that can be used to make parsers resemble a grammar very closely. However, a wide range of other combinators is also imaginable, e.g., to cover common patterns like repetition or optionality. Thereby partial results can be composed in a flexible way. Further application-specific combinators can be added easily. The non-linear structure of graphs suggests even more interesting patterns worth an abstraction.

Parser combinators are very popular, because they integrate seamlessly with the rest of the program and hence the full power of the host language can be used. Unlike Yacc [5] no extra formalism is needed to specify the grammar. Another benefit is that parsers are first-class values within the language. For example, we can construct lists of parsers or pass them as function parameters. The possibilities are only restricted by the potential of the host language.

Having all these benefits in mind the question arises how parser combinators can be adopted to graphs. The discussion of this idea is the main contribution of this paper. We introduce the theoretical background in Sect. 3 and 4. Thereafter, we propose a framework and a set of graph parser combinators in Sect. 5. Then we go on to demonstrate the practical use of these combinators by applying them to a real-world example, namely the visual language VEX (visual expressions). This example is introduced in Sect. 2 and the corresponding parser is constructed in Sect. 6. Therewith, we demonstrate that efficient special-purpose graph parsers can be implemented straightforwardly.

A First Impression

At this point, we provide a toy example to give an impression of what a parser constructed using our combinators is going to look like.

An important advantage of the combinator approach is that a more operational description of a language can be given. For example, the language of the strings $\{a^k b^k c^k | k \in \mathbb{N}\}$ is not context-free. Hence a general-purpose parser for context-free languages cannot be applied at all, although parsing this language actually is very easy: "Take as many a characters as possible, then accept exactly the same number of b characters and finally accept exactly the same number of c characters."

Using PolyParse [6], a well-known and freely-available parser combinator library for strings maintained by Wallace, a parser for this string language can be defined as shown in Fig. 1a. The type of this parser determines that there may be a user-state `s` (not used in this example), that the tokens have to be characters and that the result is a number (namely k). The combinator `many` applies a given parser multiple times, while collecting the results. If the given word is not a member of the language one of the calls of `exactly` fails.

```
abc::Parser s Char Int
abc =
 do as←many (char 'a')
    let k = length as
    exactly k (char 'b')
    exactly k (char 'c')
    return k
```

(a) String parser

```
abcG::NGrappa s Char Int
abcG n =
 do (n',as)←chain (dirEdge 'a') n
    let k = length as
    (n'',_)←exactChain k (dirEdge 'b') n'
    exactChain k (dirEdge 'c') n''
    return k
```

(b) Graph parser

Fig. 1. Parsers for the string and the graph language $a^k b^k c^k$

Note, that the given parser uses the `do`-notation, syntactic sugar Haskell [7] provides for dealing with monads. Monads in turn provide a means to simulate state in Haskell. In the context of parser combinators they are used to hide the list of unconsumed input. Otherwise all parsers in a sequence would have to pass this list as a parameter explicitly.

In order to motivate our approach to graph parser combinators we provide the graph equivalent to the previously introduced string parser `abc`. Strings generally can be represented as directed, edge-labeled graphs straightforwardly. For instance, Fig. 2 provides the graph representation of the string "aabbcc". A graph parser for this graph language can be defined using our combinators in a manner quite similar to the parser discussed above as shown in Fig. 1b. The main difference between the implementations of `abc` and `abcG` is, that we have to pass through the position, i.e., the node, we currently process.

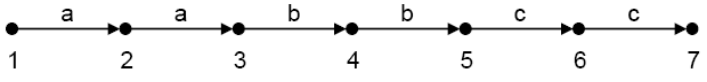
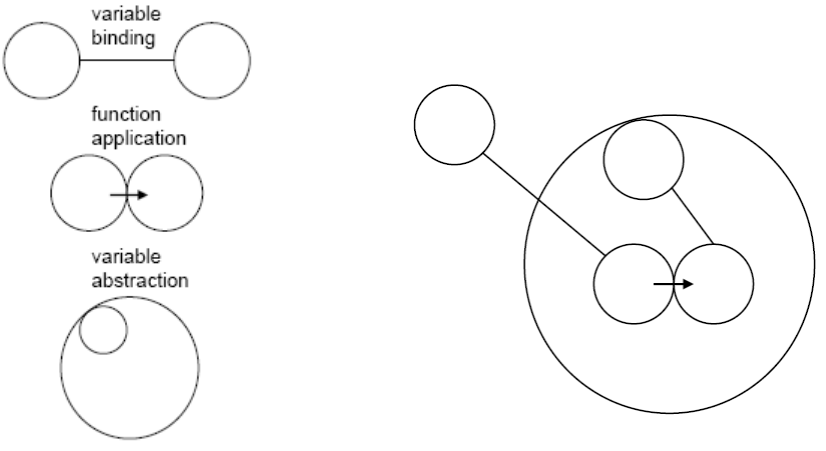


Fig. 2. The string graph "aabbcc"

Although many details have been omitted here, we hope that the idea behind graph parser combinators is understandable already. In Sect. 5 we present the framework and the combinators that can be used to conveniently implement parsers like `abcG`.

2 A Running Example: VEX

In this section we introduce the visual language VEX as our running example. Visual languages and graphs are highly related, because graphs are a very natural means for describing complex situations at an intuitive level. Graphs in particular appear to be well suited as an intermediate data structure in visual



(a) Language elements (b) Example diagram: $\lambda x.(y\ x)$

Fig. 3. The visual language VEX

language editors. For instance, in editors generated using the diagram editor generator DiaGen [1], visual objects and their spatial relations are mapped to graph components and a graph parser is used to check whether a diagram is a member of the particular language. This mapping is described in more detail in the next section.

VEX [8] is a language for the visualization of lambda expressions. Thereby variable bindings are explicitly defined by lines and not implicitly given by their names as in normal lambda calculus. In Fig. 3a the basic elements of the language are depicted. A VEX diagram basically consists of a set of circles, lines and arrows, whose layout determines the represented lambda term.

In VEX each variable identifier is represented by an empty circle (labeling text may be contained, however) that is connected by a line to a so-called root node. A root node is again an empty circle with one or more lines touching it, leading to all identifiers representing the same variable.

A root node may either be internally tangential to another circle, it then represents the parameter of a λ-abstraction, or it is not contained in any other circle, it then denotes a free variable. A circle representing a λ-abstraction contains its parameter circle and a VEX (sub-)diagram as its body.

Function application is expressed by two circles externally tangent to each other. An arrow between these circles indicates the direction of application. As usual application associates to the left. However, a different order of application can be determined by a particular numbering scheme [8].

In Fig. 3b a syntactically correct VEX diagram is given that represents the lambda term $\lambda x.(y\ x)$. We do not want to go into more detail here – [8] provides a full and more precise description of the syntax. In the following, we use the

language VEX to clarify the notions of graphs, graph grammars and parsing. Furthermore, the benefits of our actual parser combinator library are demonstrated by constructing a special-purpose parser for VEX graphs in Sect. 6.

3 Graphs

Our framework deals with a generalized notion of graphs where edges are allowed to visit an arbitrary number of nodes. Such graphs are often called hypergraphs. In this section we give a formal definition of these graphs and Haskell types for their representation.

Following [3] a graph consists of a set of edges and a set of nodes. An edge is an atomic item with a fixed number of tentacles, called the type of the edge. It can be embedded into a graph by attaching each of its tentacles to a node. Directed graphs are a special case of this notion, i.e., they are graphs whose edges are distinguished by exactly two tentacles, the first representing the source of the edge and the second the target, respectively.

Graphs are a flexible modeling concept suitable, e.g., for modeling a large number of different visual languages [9]. Using VEX as an example we briefly sketch how syntax analysis for diagrams can be performed. It is usually conducted by a chain of processing steps. For instance, in DiaGen [1] first a so-called spatial relationship graph (SRG) is created. This SRG contains a component edge for every diagram component as well as spatial relationship edges (like *contains* or *touches*) linking these components via their attachment nodes. A simple example is depicted in Fig. 4.

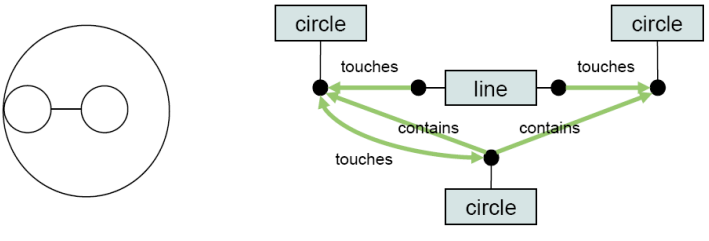


Fig. 4. Spatial relationship graph of a diagram representing the identity function

We represent edges by a rectangular box marked with a particular label and nodes by filled black circles. A line between an edge and a node indicates that the node is visited by that edge. Spatial relationship edges are directed, binary edges, so that we can represent them as colored arrows in the figure for clarity's sake.

In a second step this SRG is transformed to the reduced graph model (or abstract syntax graph) by the so-called reducer. This step is closely related to

lexical analysis in a string setting. The reduction of complexity allows more readable grammars and often reasonably efficient parsers.[1] The derivation structure (if any) yielded by the parser can finally be used for attribute evaluation and diagram beautification.

In Fig. 5 the reduced graph model of Fig. 3b is shown. Nodes additionally are labeled with their identifiers. The free variable root node in the upper left corner of Fig. 3b, e.g., is mapped to an edge labeled "freevar". The reference of the variable to this root node is mapped to an edge "bind" that links the edges "freevar" and "var".

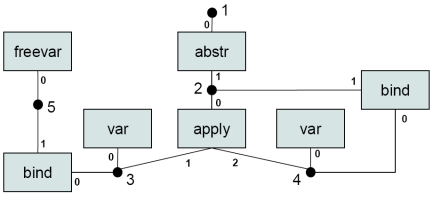


Fig. 5. Graph representation of the VEX expression of Fig. 3b

The small numbers close to the edges are the tentacle numbers. Without these numbers the image may be ambiguous, since different tentacles usually play different roles. For instance, the tentacle with number 1 of "abstr" edges always has to be attached to the body of the abstraction and the tentacles 1 and 2 of "apply" edges to a function and its argument, respectively.

Before we provide the Haskell types representing the graph data structures we have to introduce our graph model more formally. It differs from standard definitions as found in, e.g., [3] that do not introduce the notion of a *context*.

Let C be a set of labels and $type : C \to I\!N$ a typing function for C. In the following, a graph H over C is a finite set of tuples (lab, e, ns), where e is an edge identifier unique in H, $lab \in C$ is an edge label and ns is a sequence of node identifiers such that $type(lab) = |ns|$ (length of sequence). The nodes represented by the node identifiers in ns are called *incident* to edge e. We call a tuple (lab, e, ns) a *context* (in analogy to [10], although our graphs are not inductively defined).

The position of a particular node n in the sequence of nodes within the context of an edge e represents the tentacle of e that n is attached to. Hence the order of nodes matters. The same node identifier also may occur in more than one context indicating that the edges represented by those contexts are connected via this node.

Note, that our notion of graphs is slightly more restrictive than the usual one, because we cannot represent isolated nodes. In particular the nodes of H are implicitly given as the union of all nodes incident to its edges. In fact, in many graph application areas isolated nodes simply do not occur. For example, in our context of visual languages diagram components are represented by edges, and nodes just represent their connection points, i.e., each node is attached to at least one edge. As we will see, our definition is very advantageous for parsing. In particular we can transfer the concept of input consumption straightforwardly by just using contexts as tokens (cf. Sect. 5). So we ignore this issue at the moment in order to keep the technicalities simple.

[1] In case of VEX this reduction is quite complicated. We do not discuss this step here in detail.

The following Haskell code introduces the basic data structures for representing nodes, edges and graphs altogether:

```
type Node = Int
type Edge = Int
type Tentacle = Int
type Context lab = (lab, Edge, [Node])
type Graph lab = Set (Context lab)
```

For the sake of simplicity, we represent nodes and edges by integer numbers. We declare a graph as a set of contexts, where each context represents a labeled edge including its incident nodes. For instance, the VEX graph given in Fig. 5 can be represented as follows:

```
data VexLab = AbstrL | ApplyL | VarL | BindL | FreevarL

vg::Graph VexLab
vg = {(AbstrL,0,[1,2]), (ApplyL,1,[2,3,4]), (VarL,2,[4]),
      (BindL,3,[4,2]), (VarL,4,[3]), (BindL,5,[3,5]), (FreevarL,6,[5])}
```

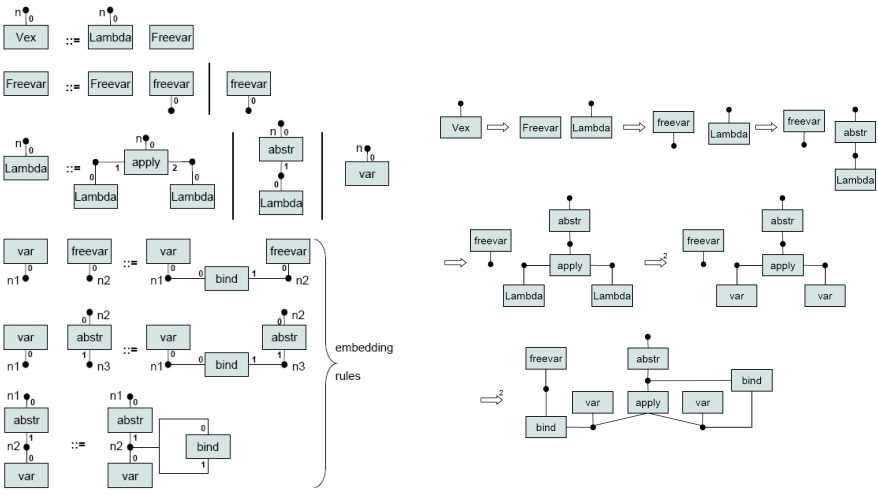
We enumerate the terminal edge labels explicitly as data literals. Using strings also is perfectly possible, but following our approach the type checker already excludes some meaningless input. The node numbers occurring in `vg` correspond directly to the identifiers given in the figure. The edges are numbered uniquely.

4 Graph Grammars and Parsers

A set of graphs, i.e., a graph language, can be defined using a graph grammar – an extension of formal language theory to graphs. A widely known kind of graph grammar are hyperedge replacement grammars (HRG) as described in [3]. Here, a nonterminal edge of a given graph is replaced by a new graph that is glued to the remaining graph by fusing particular nodes. Formally, such a HRG G is a quadruple $G = (N, T, P, S)$ that consists of a set of nonterminals $N \subset C$ (i.e., labels), a set of terminals $T \subset C$ with $T \cap N = \emptyset$, a finite set of productions P and a start symbol $S \in N$. The productions are context-free, i.e., they describe how a single nonterminal edge can be replaced with a graph.

Unfortunately this notion is not powerful enough to describe VEX; we have to extend it with so-called embedding rules – special productions that insert an additional edge into a particular context. Such context-sensitive embedding rules are needed for the description of many graph languages.

The graph grammar for VEX can be defined as $G_V = (N_V, T_V, P_V, \mathit{Vex})$ where $N_V = \{\mathit{Vex}, \mathit{Lambda}, \mathit{Freevar}\}$, $T_V = \{\mathit{freevar}, \mathit{apply}, \mathit{abstr}, \mathit{var}, \mathit{bind}\}$ and P_V contains the productions given in Fig. 6a. By convention we use lower case letters to label terminal edges; in contrast labels of nonterminal edges start with a capital letter. The productions are written very similar to BNF known from string grammars, i.e., left-hand side *lhs* and right-hand side *rhs* are separated by the symbol `::=` and several *rhs* of one and the same *lhs* can be combined by vertical bars. Node labels are used to identify corresponding nodes of *lhs*



(a) Productions of the grammar (b) Derivation of the graph given in Fig. 5

Fig. 6. The graph language of VEX

and *rhs*. They act as variables. In order to apply a production they have to be instantiated with nodes actually occurring in the graph.

As usual the language defined by a grammar is given by all terminal graphs derivable in an arbitrary number of steps from the start symbol. Note again, that the grammar of VEX is not context-free, i.e., there are productions with more than one edge at their *lhs*, although the graph language is quite simple. In particular the embedding of the "bind" edge cannot be defined in a context-free manner. Thus, the derivation is not a tree, but a DAG. The given grammar also does not ensure that there has to be exactly one "bind" edge connected to each variable. Such restrictions usually have to be expressed by so-called application conditions (see, e.g., [1]). A derivation of our example VEX graph (Fig. 5) is given in Fig. 6b.

A general-purpose graph parser for HRGs gets passed a particular HRG and a graph as parameters and constructs a derivation tree of this graph according to the grammar. This can be done, for instance, in a way similar to the well-known algorithm of Cocke, Younger and Kasami [2] in the context of strings (all HRGs can be transformed to the graph equivalent of the string notion Chomsky Normal Form). However, such a parser could not deal with our VEX grammar, because of the embedding rules.

Another problem regarding this grammar is caused by the "Freevar" productions. Since the terminal "freevar" edges are not connected in a specific way a lot of different derivations are possible. It is simply not clear, in which order the different "freevar" edges have to be derived (although their actual order does not matter at all). A general-purpose parser usually performs poorly on such highly ambiguous grammars.

Having this in mind, we can state, that a general-purpose parser capable of parsing VEX graphs has to be quite powerful. Hence the language VEX strongly motivates the construction of a special-purpose parser.

5 A Graph Parser Combinator Library

The main contribution of this paper is the proposal of the combinator approach to graph parsing. To validate this concept we provide a prototypical implementation of a respective library in this section. We later demonstrate that therewith the construction of special-purpose parsers can be greatly simplified.

Our design goals have been:

- Intuitive look and feel, i.e., short training period for people already familiar with parser combinators.
- Straightforward translation of a grammar to a parser.
- Simple parsers for simple languages even if the grammar of the language is complicated.[2]
- Sufficient performance for practically relevant applications.

In implementing our parsers we do not start from scratch. Rather we use PolyParse [6] as a base: a light-weight monadic parser combinator library already mentioned in the introduction. PolyParse has in particular appeared to be well-suited for practical applications. Thus, our approach has the additional benefit that many users are already familiar with the usage of the proposed framework.

However, we have to adapt the library a little to deal with sets of tokens rather than lists. The new type `Parser` can be defined as follows:

```
newtype Parser s t a = P (s → Set t → (EitherE Error a, s, Set t))
```

The type parameter `t` defines the type of the tokens. As a witness of success the parser returns a value of type `a`. In addition to the set of tokens, a state of type `s` is carried along. For instance, we will need such a state when parsing VEX to keep track of the root nodes in scope. `EitherE` is a type similar to `Either` that additionally provides support for different gradations of failing.

The existing instance of the type class `Monad` for the type `Parser` can be reused without any changes. Thus, we also

- hide the remaining input,
- keep track of an explicit state, which can be updated while parsing,
- allow further parsers to be parameterized with the results of previous ones (cf. `abcG`),
- support the convenient construction of results with `return`.

However, we cannot reuse several primitive parsers from PolyParse (mainly `next::Parser s t t`) since they depend on the first token of the input list,

[2] A good example is the language of the string graphs $\{a^k b^k c^k | k \in I\!N\}$. In contrast to the string language there is a context-free graph grammar describing this language, however, it is quite complicated despite the simplicity of the language (cf. [3]).

Table 1. Set-specific primitive parsers and combinators

Name	Type	Description
`aSatisfying`	`(t->Bool)-> Parser s t t`	a token satisfying a particular condition, nondeterministic and consuming
`remainingInp`	`Parser s t (Set t)`	the whole remaining input, not consuming
`satisfyingInp`	`(t->Bool)-> Parser s t (Set t)`	all tokens satisfying a particular condition, consuming
`oneOf`	`[Parser s t a]-> Parser s t a`	chooses the first parser in the list that succeeds
`best`	`[Parser s t a]-> Parser s t a`	chooses the parser consuming the largest part of the input successfully
`eoi`	`Parser s t ()`	for complete input consumption, otherwise a correct subgraph is extracted

which is meaningless in the context of sets. Instead we have to add several set-specific primitive parsers listed in Table 1 (we omit their declarations here, they are rather technical). The use of the also listed combinator `best` will become clear shortly.

Note, that up to now we have not defined any graph-specific combinators. Indeed our basic framework can be used to parse all kinds of token sets.

After this preparatory work we can define graph-specific combinators quite conveniently. However, let us first define our basic graph parser type by specializing the type `Parser`. Thereby we choose contexts (as introduced in Sect. 3) as our token type. The name `Grappa` is an acronym for graph parsing:

```
type Grappa s lab a = Parser s (Context lab) a
```

A main difference between graph parsing and string parsing is that in a graph setting we normally do not know where to actually start parsing. There is no such thing as a first token/context. Thus, most of our parsers will need a start node as a parameter explicitly. For convenience, we define an additional type `NGrappa` that hides this node parameter:

```
type NGrappa s lab a = Node→Grappa s lab a
```

Most combinators will return a result of type `NGrappa`. The following function converts such an `NGrappa` to a normal `Grappa` by trying every node of the graph as a starting point successively.

```
nGrappaToGrappa::NGrappa s lab a→Grappa s lab a
nGrappaToGrappa ng = do
                     g←remainingInp
                     best (map ng (nodes g))
```

Using `remainingInp` the current set of tokens, i.e., graph, is queried (but not consumed). Thereafter the combinator `best` is used to find the best starting

node, i.e., the one the largest part of the input can be successfully consumed from.[3] We construct the list of possible parsers by mapping `ng` to all nodes of the graph. Of course this function has to be used with care – at least if performance is an issue. However, its declaration demonstrates how flexibly parser combinators can be composed.

We further provide a combinator `oneOfN` for dealing with alternatives based on type `NGrappa`.

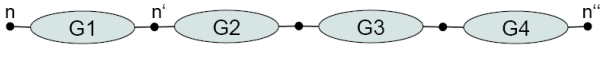
```
oneOfN::[NGrappa s lab a]→NGrappa s lab a
oneOfN ngs n = oneOf (map ($n) ngs)
```

In Table 2 we briefly sketch some of the graph-specific primitive parsers provided by our library. They are all nondeterministic and consume the context they return. The parser `dirEdge` from the introductory example (Fig. 1b) is a specialization of `edge` that uses 0 as the incoming and 1 as the outgoing tentacle number, respectively. Those numbers correspond to source and target of a binary edge.

Table 2. Some graph-specific primitive parsers

Name	Type	Description
`context`	`(Context l->Bool)->` `Grappa s l (Context l)`	a context satisfying a particular condition
`labContext`	`Eq l=>l->` `Grappa s l (Context l)`	a context with a particular label
`edge`	`Eq l=>l->Tentacle->Tentacle->` `NGrappa s l (Node,Context l)`	a labeled context connected to the active node via a particular tentacle also returning its successor (via the other, outgoing tentacle)
`connLab-` `Context`	`Eq l=>l->[(Tentacle,Node)]->` `Grappa s l (Context l)`	a labeled context connected to the given nodes via the given tentacles

We omit the implementation details of the primitive parsers and rather switch the focus to combinators. A first important combinator is `chain`, in a sense the graph equivalent of `many`. It can be used to parse several successive graphs connected via intermediary nodes as shown in the figure below.



Therefore, a given `NGrappa` is applied as long as possible assuring proper connections between the different parses. As a result the list of the partial results

[3] This is an expensive operation. However, the alternative of returning an arbitrary result is often not meaningful in a graph setting. For instance, the parser `abcG` would always succeed immediately returning $k = 0$.

is returned. The connecting nodes are especially important, because they have to be provided as starting points for the parser one at a time. This is realized by passing through the active node, i.e., the parser has to also return the node where it stops.

```
chain::NGrappa s lab (Node,a)→NGrappa s lab (Node,[a])
chain p n = do {
            (n',x)←p n;
            (n'',xs)←chain p n';
            return (n'',x:xs)
            } `onFail` return (n,[])
```

The string combinator `exactly` used in the introduction can be carried over to graphs quite easily, too:

```
exactChain::Int→NGrappa s lab (Node,a)→NGrappa s lab (Node,[a])
exactChain 0 p n = return (n,[])
exactChain num p n = do
                     (n',x)←p n
                     (n'',xs)←exactChain (num-1) p n'
                     return (n'',x:xs)
```

Note, that with `chain` and `exactChain` all combinators used in the definition of our motivating example `abcG`, i.e., the parser for the string graphs $\{a^k b^k c^k | k \in \mathbb{N}\}$, are introduced already.

In addition to these combinators several parser combinators known from string parsing can be reused for graph parsing straightforwardly, e.g., `oneOf`. Furthermore, there are some combinators that have to be used with care. Their semantics changes in a graph setting, because they do not maintain proper connections (unlike strings the order of tokens, i.e., contexts, does not represent a particular kind of connection anymore). For instance, the combinator `many` can be used in the context of graphs just to parse more or less independent subgraphs or star shapes (when dealing with `NGrappa`s):

```
star::NGrappa s lab a→NGrappa s lab [a]
star ng = many ∘ ng
```

Note, that several combinators and associated parsers have to pass through the active node. In particular parsers for languages of string graphs, therefore, include some boilerplate code (cf. Fig. 1b). It is common practice to hide this complexity in an additional state monad. However, practical graph languages usually are more branched. Due to this property the monadic approach results in frequent updates of this state. Thus, we prefer passing the active node as a function parameter explicitly.

There are a lot more combinators imaginable. Our library, e.g., provides further combinators for dealing with trees, separators, etc. However, the ones presented in this section give a good first insight. Furthermore, they are in particular needed to tackle our example VEX.

6 Parsing VEX

In this section we construct a parser for the example language VEX using the combinators defined in the previous section. The purpose of the presentation of this parser is twofold. First, we demonstrate how the combinators have to be used. And second, we intend to show that special-purpose graph parsers can be defined quite straightforwardly.

Our goal is to map a VEX graph to the λ-term it represents. Hence, we define the type `Lambda` as the result type of the parser.

```
data Lambda = Abstr String Lambda |
              Apply Lambda Lambda |
              Var String
```

Note, that `Lambda` is a kind of tree and not a DAG as one might expect, because variable bindings are resolved by proper naming. Here an additional advantage of parser combinators shows up. We can compute a useful result while parsing and need not post-process an intermediate data structure somehow representing the derivation.

VEX is an example where we really need a parser state. In particular we store a number used to construct a name for the next fresh variable to be bound in an abstraction. Moreover, the state contains a lookup table that maps node numbers to variable names. The type `VexState` represents these requirements.

```
type VexState = (Int,[(Node,String)])
```

The top-level parser `vex` first consumes all (`many`) "freevar" edges using the parser `freevar`. Thereby the lookup table is extended properly introducing fresh names for the new variables. We are not interested in the parsing order of the "freevar" edges. Hence, we `commit` the intermediate results to prevent backtracking. Finally, the remaining graph is parsed using the parser `lambda`.

```
vex::Grappa VexState VexLab Lambda
vex = do
      many (commit freevar)
      nGrappaToGrappa lambda

freevar::Grappa VexState VexLab ()
freevar = do
          (_,_,[n])←labContext FreevarL
          stUpdate (λ(nn,vt)→(nn+1,(n,"v"++show nn):vt))
```

The parser `lambda` accepts either an application (`apply`), an abstraction (`abstr`) or a variable (`var`). All of these subordinated parsers have the same type.

```
lambda,apply,abstr,var::NGrappa VexState VexLab Lambda
lambda = oneOfN [apply, abstr, var]
```

At "apply" edges there are two directions to further process the graph. Both have to be traversed and must yield a valid subgraph. This is a main difference

to string parsing where the next token is always predetermined. We deal with this issue by parsing an application depth-first beginning with its left side. Since the diagram language VEX does not support sharing of common subexpressions we can consume recognized input unconditionally anyhow.

```
apply n = do
          (_,_,ns)←connLabContext ApplyL [(0, n)]
          l1←lambda (ns!!1)
          l2←lambda (ns!!2)
          return (Apply l1 l2)
```

The nodes visited via particular tentacles (here, 1 and 2) are accessed by using the tentacle number as a parameter to the list index operator `(!!)`. Both subgraphs have to be parseable with `lambda` again. The overall result then is composed by the application of the data constructor `Apply` to the results yielded by parsing these subgraphs.

Parsing an abstraction means introducing a new variable in the lookup table that can be released after parsing its body. The state can be queried and changed using `stGet::Parser s t s` and `stUpdate::(s->s)->Parser s t ()`, respectively. These functions are provided by PolyParse for dealing with the user-state.

```
abstr n = do
          (_,_,ns)←connLabContext AbstrL [(0, n)]
          oldstate@(nn,vt)←stGet
          stUpdate (const (nn+1,(ns!!1,"v"++show nn):vt))
          l←lambda (ns!!1)
          stUpdate (const oldstate)
          return (Abstr ("v"++show nn) l)
```

Edges labeled “var” represent variables. Additionally, there has to be a “bind” edge to a node that can be mapped to a variable name via the lookup table. Note, that it would even be possible to raise the error level, i.e., prevent backtracking, if there is no corresponding “bind” edge. However, we still backtrack, since we are also interested in finding correct subgraphs.

```
var n = do
        connLabContext VarL [(0, n)]
        (_,_,ns)←connLabContext BindL [(0, n)]
        (_,vt)←stGet
        case lookup (ns!!1) vt of
          Nothing→fail $ show (ns!!1) ++
                              " out of scope!"
          Just v→return (Var v)
```

The parser for VEX graphs can be called using the function `vexParse` that applies the PolyParse function `runParser` to the initial state `(1,[])` and just returns the result or an error message.

```
vexParse::Graph VexLab→Either Error Lambda
vexParse = getResult ∘ runParser vex (1,[])
```

For our example graph `vg` it succeeds with the result `λv2→(v1 v2)`.

A Brief Remark on Performance

In [3] it is proved that there are (even context-free) graph languages for which parsing is NP-complete. These languages, of course, cannot be parsed with our combinators efficiently. However, there are also languages on which most general-purpose graph parsers perform poorly, although they can be parsed efficiently.

For instance, even the quite simple language VEX causes a general-purpose parser to run in exponential time (at least, if it is not tweaked somehow). The exponent thereby depends on the number of "freevar" edges, since those can be derived in a lot of different orders ($n!$). The parser presented previously, however, is not affected by this issue: We simply have committed to the first result at any one time. Similar language-specific performance optimizations can be applied to many languages.

We have also measured the execution time to parse the string graphs $a^k b^k c^k$ for several k using a general-purpose parser (also implemented in Haskell). As already mentioned it is possible to describe this graph language with a HRG. It turns out that these string graphs can be parsed in polynomial time regarding this grammar, however, performance is worse than one might expect for such a simple language.

In contrast, the special-purpose parser presented as a motivating example in the introduction can be applied to very large graphs. Even if we do not know the starting node and all nodes have to be tried successively we only get a factor of $3*k$. This still outperforms our general-purpose parser. String graphs are a rather artificial example, though. We have to conduct a more elaborate performance comparison in the future.

7 Related Work

To our best knowledge graph parser combinators have not been considered up to now. So in this section we briefly sketch several related approaches to parsing in general and dealing with graphs in functional languages.

Besides PolyParse [6] there are other parser combinator libraries that are also widely-used. For instance, Parsec [11] is well-known for its high performance and good error reporting capabilities. The main difference between Parsec and PolyParse is that Parsec is predictive by default only backtracking where an explicit `try` is inserted whereas in PolyParse backtracking is the default except where explicitly disallowed by a `commit`. We have to gain more experience with graph parser combinators to judge which approach is better suited in our setting.

In the context of strings the complement of the parser combinator approach is parser generation. Thereby a grammar is given, e.g., in EBNF from which a real parser in a particular language can be generated. The tool Happy [12] can be used to generate such a Haskell parser. However, the advantage of a parser generator for strings – namely that efficient parsers can be generated for nearly arbitrary context-free grammars – does not count that much in a graph setting.

At this point we also have to mention the work of Erwig [10], because our declarations are mainly inspired by his approach. Erwig criticized the imperative

style of the algorithms described in, e.g., [13] and proposed a new approach: looking at graphs as inductively defined data types. So he presented a graph declaration where nodes are added inductively one at a time. Incident edges are represented as a part of their so-called contexts. Unfortunately his library [14] does not generalize to hypergraphs and also does not support graph rewriting and parsing.[4]

Also highly related are approaches that aim at the combination of functional programming and graph transformation. At the time of writing a textbook is work in progress that provides an implementation of the categorical approach to graph transformation with Haskell [16]. Since graphs are a category a higher level of abstraction is used to implement graph transformation algorithms. An even more general framework is provided in [17]. The benefit of this approach is its generality since it just depends on categories with certain properties. However, up to now parsers are not considered, so we cannot compare usability and performance.

8 Conclusion and Further Work

In this paper we have introduced graph parser combinators, a novel approach to the construction of special-purpose graph parsers. By applying the combinator approach to the domain of graph parsing we further have demonstrated that this well-known technique can be used to parse all kinds of non-linear structures.

Graph parsing is highly relevant for real world applications as we have demonstrated with an example from the domain of visual languages. Here, graph parsing is crucial for the syntactical analysis of diagrams.

Further we have demonstrated that our library can be flexibly used and easily extended. It allows the construction of graph parsers even for context-sensitive graph grammars. Since the implementation of our combinators is quite similar to the more conventional string-based parser combinator libraries Haskell programmers will be familiar with its usage immediately. In combination with the implemented general-purpose graph parser (not discussed in this paper) we have taken an important first step towards a functional graph parsing library.

As we have backed up by examples our library in its present form is perfectly usable already. Nevertheless a lot of work remains to be done. First of all, we have to provide a solid theoretical foundation. We particularly have to investigate how to simplify the process of writing correct parsers using our framework. Therefore, we need to provide mappings from graph grammar formalisms to parsers on top of our framework. This would permit the more precise evaluation of graph parser combinators with respect to power and performance.

Another area of future research is the extension of the set of combinators. Compared to strings graphs can exhibit many more interesting patterns. And finally, it would be interesting to see how capabilities for error recovery could be

[4] In fact, Erwig discussed termgraph rewriting in [15], however, the presented algorithm is tailored to the problem and cannot be generalized straightforwardly.

added (like [18] for strings) or how a breadth-first search strategy would affect the performance.

All in all, we propose the following as an attractive approach to graph parsing: First, try using a general-purpose graph parser. If it is applicable and reasonably efficient everything is ok. However, if additional flexibility is needed or performance is an issue consider the use of graph parser combinators. The library presented in this paper permits the rapid construction of special-purpose graph parsers. Thereby, language-specific performance optimizations can be incorporated easily.

Acknowledgements

We would like to thank Malcolm Wallace for his useful comments regarding PolyParse, Martin Erwig, whose work on graphs in a functional setting inspired this paper, and the anonymous referees for a lot of valuable suggestions.

References

1. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (2002)
2. Kasami, T.: An efficient recognition and syntax analysis algorithm for context free languages. Scientific Report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachussetts (1965)
3. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. I, pp. 95–162. World Scientific, Singapore (1997)
4. Hutton, G., Meijer, E.: Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996)
5. Johnson, S.C.: Yacc: Yet another compiler compiler. Technical Report 32, Bell Laboratories, Murray Hill, New Jersey (1975)
6. Wallace, M.: PolyParse (2007), `http://www.cs.york.ac.uk/fp/polyparse/`
7. Peyton Jones, S.: Haskell 98 Language and Libraries. The Revised Report. Cambridge University Press (2003)
8. Citrin, W., Hall, R., Zorn, B.: Programming with visual expressions. In: Haarslev, V. (ed.) Proc. 11th IEEE Symp. Vis. Lang, pp. 294–301. IEEE Computer Soc. Press, Los Alamitos (1995)
9. Minas, M.: Hypergraphs as a uniform diagram representation model. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 281–295. Springer, Heidelberg (2000)
10. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. 11(5), 467–492 (2001)
11. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Dept. of Comp. Science, Universiteit Utrecht (2001)
12. Gill, A., Marlow, S.: Happy - the parser generator for Haskell, `http://www.haskell.org/happy`
13. King, D.: Functional Programming and Graph Algorithms. PhD thesis, University of Glasgow (1996)

14. Erwig, M.: FGL - A Functional Graph Library, http://web.engr.oregonstate.edu/~erwig/fgl/haskell/
15. Erwig, M.: A functional homage to graph reduction. Technical Report 239, FernUniversität Hagen (1998)
16. Schneider, H.J.: Graph transformations - an introduction to the categorical approach (2007), http://www2.cs.fau.de/~schneide/gtbook/
17. Kahl, W., Schmidt, G.: Exploring (finite) Relation Algebras using Tools written in Haskell. Technical Report 2000-02, Fakultät für Informatik, Universität der Bundeswehr, München (2000)
18. Swierstra, S.D., Azero Alcocer, P.R.: Fast, error correcting parser combinators: a short tutorial. In: Pavelka, J., Tel, G., Bartosek, M. (eds.) SOFSEM 1999. LNCS, vol. 1725, pp. 111–129. Springer, Heidelberg (1999)

Testing Erlang Refactorings with QuickCheck

Huiqing Li and Simon Thompson

Computing Laboratory, University of Kent, UK
{H.Li,S.J.Thompson}@kent.ac.uk

Abstract. Refactoring is a technique for improving the design of existing programs without changing their behaviour. Wrangler is a tool built at the University of Kent to support Erlang program refactoring; the Wrangler tool is written in Erlang.

In this paper we present the use of a novel testing tool, Quviq QuickCheck, for testing the implementation of Wrangler. QuickCheck is a specification-based testing tool for Erlang. With QuickCheck, programs are tested by writing properties in a restricted logic, and using the tool these properties are tested in randomly generated test cases.

This paper first gives overviews of Wrangler and Quviq QuickCheck, then discusses the various ways in which refactorings can be validated, and finally shows how QuickCheck can be used to test the correctness of refactorings in an efficient way.

1 Introduction

Refactoring [7] is a technique for transforming program source code in such a way that it changes the program's internal structure and organisation, but not external behaviour. The key characteristic that distinguishes refactoring from general code manipulation is its focus on structural change, strictly separated from changes in functionality. Functionality-preservation requires that refactorings do not introduce (or remove) any bugs. Refactorings typically have two aspects: *program analysis* is required to check that certain side-conditions are met by the program in question in order for the refactoring to preserve behaviour, and *program transformation* which carries out the actual program restructuring. In a slogan: *"Refactoring = Condition + Transformation"*.

Refactorings can be done manually, but this can be tedious and error-prone for small programs, and impractical for larger systems. Software tools (*"refactoring engines"*) can help programmers perform refactorings automatically, and are available for a variety of languages, including Smalltalk, Java, C#, C++, Haskell, Erlang, etc. With a refactoring tool, the programmer only needs to select which part of the program to be refactored and which refactoring to apply, and the tool will automatically check the side-conditions and apply the transformation throughout the whole program if the side-conditions are satisfied. Wrangler is the tool that we are implementing to support refactoring Erlang [1] programs, and this forms one aspect of 'Formally-Based Tool Support for Erlang Development'[1] [6], a joint research project between Universities of Kent and Sheffield.

[1] FORSE is supported by EPSRC, UK.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 19–36, 2008.

Implementing a practical and usable refactoring tool for a real world programming language is by no means trivial. A refactoring tool needs to get access to the program's syntax and static semantics (possibly including type information), to implement different kinds of program analysis and transformation, and to preserve the comments, and potentially, layout, of the transformed program. Among other criteria, such as efficiency, usability and completeness, the reliability of a refactoring tool is vital for it to be accepted in practice. A bug within a refactoring tool can introduce bugs in the refactored programs silently, and such bugs may be impossible to detect statically, if they result in a valid program which behaves differently from the original.

The correctness of refactorings implemented can be ensured from several aspects including, but not limited to, a clear specification clarifying the preconditions, transformation rules, and/or post-conditions of each refactoring; a verification that argues the correctness of the specification, and most importantly a thorough testing of the refactoring tool. A traditional way of testing refactoring tools is to create test cases manually. Each test case contains an input program, a refactoring command, and the expected result, which could be either the refactored version of the input program or the original input program (along with a failure message) depending on whether the side-conditions are satisfied. Then these tested cases are usually run with a unit testing tool, such as EUnit [3] for Erlang. Writing test cases manually is tedious and hard to cover all possible refactoring scenarios. Incomplete test suite potentially leaves bugs in refactoring tools.

We present the technique of using Quviq QuickCheck [8], a tool developed by Quviq AB, to automate the testing of Wrangler. Instead of writing small test programs, we use real-world available Erlang programs as our refactoring input programs. For example, one of the Erlang programs we have used is Wrangler itself, which currently contains 25 modules, 20K lines of code in total. Quviq QuickCheck tests running code against formal specification, using controllable random test case generation combined with automated test case simplification to assist error diagnosis. With Quviq QuickCheck, we automate the generation of refactoring commands and the checking of refactoring outputs. Refactoring commands are generated randomly using the information stored in the annotated abstract syntax tree (AAST) of the input program. Along with the development of each refactoring, we write a collection of properties that the refactoring should satisfy. Failing to meet one or more of these properties indicates bugs in the implementation or properties. Each time the testing is run, it generates 100 refactoring commands by default, applies each command to the input program, and checks that the properties being tested return true in every case. This way, we are able to integrate the specification and testing of refactorings very naturally.

The rest of the paper is structured as follows. In sections 2 and 3, we give introductions to Wrangler and Quviq QuickCheck. Section 4 gives an overview of the different ways in which refactoring engines can be tested, and in section 5 we explain our approach to testing Wrangler with QuickCheck, including the

generation of refactoring commands, and the kind of general properties that we use to test refactorings. In section 6, as an example, we illustrate the testing of *renaming a function*. In section 7, we give an evaluation of our approach; related work is presented in section 8, and conclusions and future work are given in section 9.

2 Wrangler – An Erlang Refactorer

Wrangler [11,12] is the tool that we are building to provide support for interactive refactoring of Erlang programs. The current version of Wrangler supports a number of structural refactorings, including *rename an identifier*, *generalise a function definition*, *function extraction*, *move a function definition between modules*, *fold expressions against a function definition*, etc, and functionalities for duplicated code detection. More process structure related refactorings are being added.

Wrangler is built on top of the Erlang `syntax-tools` package [14] which provides a representation of the Erlang AST within Erlang. `syntax-tools` allows syntax trees to be augmented with additional information as necessary. The Wrangler AST representation is annotated with a variety of information:

- Comments in the source code are inserted as attachments to the nodes in the AST at the appropriate place.
- Each function or variable name is associated with its actual source location and the location of its defining occurrence, thus reflecting the binding structure of the program.
- The start and end location of each syntactic entity in the source code is also stored in the augmented AST, allowing entities to be located by means of their position, as well as supporting pretty-printing facilities.
- Category information indicating the kind of syntax phrase the AST node represents, such as expression, function, pattern and so on is also included in the tree.
- Finally, free and bound variable information is also attached to the AST representation of each syntax phrase in the source code.

Wrangler is embedded in the Emacs editing environment; to manage communication between the refactoring engine and Emacs we make use of the functionalities provided by Distel [13], an Emacs-based user interface toolkit for Erlang.

To perform a refactoring with Wrangler, the focus of refactoring interest has to be selected in the editor first. For instance, an identifier is selected by placing the cursor at any of its occurrences; an expression is selected by highlighting it with the cursor. Next, the user chooses the refactoring command from the *refactor* menu, and inputs the parameter(s) in the mini-buffer if required. The Wrangler tool checks that the focus item is suitable for the refactoring selected, that the parameters are valid, and that the refactoring's side-conditions are satisfied.

If all these checks are successful, Wrangler will then perform the refactoring, and update the program with the new result, otherwise it will give an error

message, and abort the refactoring with the input program unchanged. *Undo* is supported by Wrangler; applying *undo* once reverts the program back to its state immediately before the last refactoring was performed.

Snapshots of Wrangler are given in Figures 1-2 with a particular refactoring scenario showing the generation of function `repeat/1`. In Figure 1, the user has selected the expression `io:format("Hello\n")` in the definition of `repeat/1`, has chosen the *Generalise Function Definition* command from the

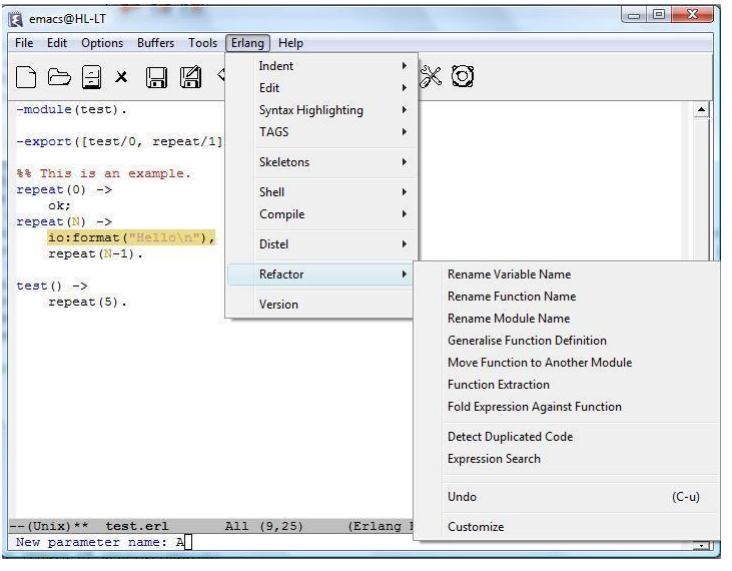


Fig. 1. A snapshot of Wrangler

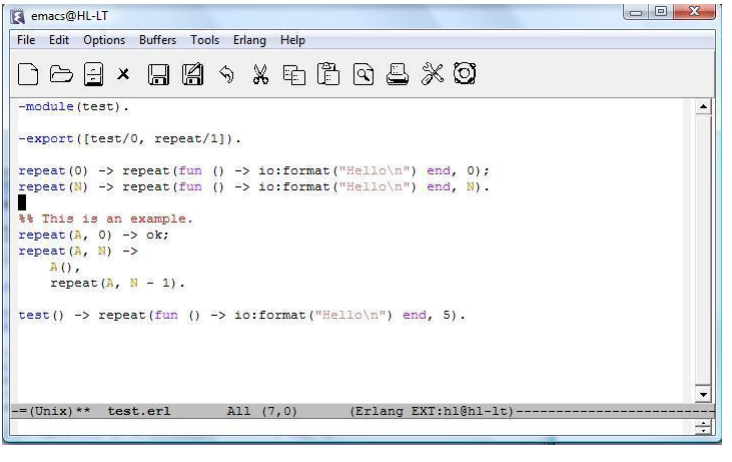


Fig. 2. A snapshot of Wrangler showing the result of generalisation

Refactor menu, and is just entering a new parameter name A in the mini-buffer. Then the user would press the *Enter* Key to perform the refactoring. After the side-condition checking and program transformation, the result of this refactoring is shown in Figure 2: the new parameter A has been added to the enclosing function definition `repeat/1`, which now becomes `repeat/2`; the highlighted expression has been replaced with `A()`; and at the call-site of the generalised function, the selected expression, wrapped in a *fun*-expression, is now supplied to the function call as its first actual parameter. We enclose the selected expression within a function closure because of its side-effect, so as to ensure that the expression is evaluated at the proper points. As a design decision, if the generalised function is exported by the current module, an auxiliary function is created to ensure that the interface of the module is unchanged, as shown in this example.

3 Quviq QuickCheck

Quviq QuickCheck is a property-based testing tool, developed from Claessen and Hughes' earlier QuickCheck tool for Haskell [4], re-designed for Erlang with a number of extensions, of which the most significant is an ability to simplify failing test cases automatically [8].

Quviq QuickCheck provides an API in Erlang that allows users to write *properties* that are expected to hold of programs; these properties are themselves expressed as Erlang source code. QuickCheck also defines a variety of *generators* and combining forms for generators by means of which the user can generate test data of the appropriate type and distribution for their needs.

As an example, consider the standard list reverse function. One property of this function is expressed thus:

```
prop_reverse() -> ?FORALL(Xs, list(int()),
                          list:reverse(list:reverse(Xs))== Xs).
```

As an abstract property, this says that reversing a list of integers twice has the result of returning the original list. In QuickCheck, the functions `int/0` and `list/1` are both data generators: `int/0` generates random integers, and `list/1` generates a list of elements generated by its argument. `?FORALL` is an Erlang macro. `?FORALL(X, Gen, Prop)` binds `X` to a value generated by `Gen` within the property `Prop`. The example property will be said to hold in QuickCheck if `list:reverse(list:reverse(Xs))== Xs)` holds for all values of `Xs` generated by `list(int())`.

The property is checked by running 100 random test cases generated by the generators, and reports success if all tests pass this. If any test case fails, the (first such) failing case will be printed. 100 is the default value of the number of test cases generated in each run of QuickCheck, and this figure can be customised by the user.

A failing test case indicates bugs in either the implementation under test or the written properties. For example, testing the following property

```
prop_list_delete()->
    ?FORALL(I, int(),
        ?FORALL(List, List(int()),
            not (lists:member(I, lists:delete(I, List)))))
```

against the standard function `lists:delete/2` might report

```
Failed! After 37 tests.
-8
[5, -8, 12, -8, 9]
```

as `lists:delete(I, List)` only removes the first occurrence of I in List. Once a counterexample has been found, the *shrinking* functionality provided by Quviq QuickCheck will allow QuickCheck to minimise the failing case as much as possible. For the above example, the length of the counterexample data will be reduced, and the output above would be augmented by

```
Shrinking......(6 times)
-8
[-8,-8]
```

By writing properties in this style, a QuickCheck user can build up a formal specification, which is then checked against the implementation by QuickCheck. The mutual testing of implementation and specification ensures the correctness of both.

In comparison with traditional automated testing, as provided by systems such as EUnit [3], which runs the same set of tests repeatedly, QuickCheck allows the user to run many different tests with little effort, therefore has the potential to find more bugs. It is, of course, possible to re-run tests simply by re-using a seed value within the random generation, and so to ensure that regression testing takes place if required.

The API provided by QuickCheck contains functions for generating both simple and complex test data, according to distributions described by the user, as well as macros for writing and testing properties. In the following sections, an explanation will be given when an API function or macro is used.

4 Validating Refactoring Engines

Refactorings and refactoring engines can be validated in a number of different ways. In this section we present an overview of the various approaches and their pros and cons, before explaining our approach in more detail in the next section.

In checking whether the result of a refactoring has preserved behaviour, the result naturally needs to compile and run without errors; in the remainder of this section we assume that the results are also checked for being compilable as well as being tested in various ways we discuss.

4.1 Regression Testing of Refactored Programs

The most popular means of validating refactorings in current use is to ensure that refactored code meets all the tests that the original version met. As Object-Oriented refactoring has been identified as one of the central characteristics of an extreme programming approach, it is reasonable to assume that the test data will already be in place, and so the advantage of this approach is that the cost of testing the refactored code is small. This approach means that the refactored code has the same warranty as the original code.

The approach has two limitations. First, the coverage of the code is necessarily partial, and so it is possible that bugs have been introduced in the untested parts of the code. Also, the testing cost can be higher in cases where the test cases have themselves to be refactored: for instance, if a function is generalised, then it is necessary to add an extra datum to the test data for each function call.

4.2 Testing the Old and New Programs

A variant of the previous approach tests the two versions of the program against each other: on input data taken from an existing test suite, the outputs from two versions of the program can be compared directly. This approach is lower cost in the case where there is no pre-existing test data, since it is not necessary explicitly to state the output values corresponding to the various input data. A disadvantage is that any framework needs to accommodate the co-existence of two versions of the code under test.

Neither this nor the previous approach actually checks the structural changes of the refactored code, and could fail to test that refactorings actually achieve their purpose. For example, program behaviour preservation can be achieved even if a malfunctioning refactoring returns the program structurally unchanged without giving an error message.

4.3 Programs as Data

In contrast to the earlier approaches, it is possible to see the refactoring as a program, and so to supply it with a set of input programs and the corresponding output programs that are expect to result. Two variants of the check are possible:

- It is possible to analyse the abstract syntax tree (AST) resulting from the transformation, and to compare this with the expected result. This neglects the layout of the refactored program.
- In contrast, it is possible to specify the source code to be expected, with a given program layout. This is a stronger test than the former; since it not only prescribes the AST but also its particular layout, but this approach is appropriate when refactoring code is expected to be laid out in a way that will make it recognisable to its author.

This was the approach that we used first, using the Haskell package HUnit for testing HaRe (the Haskell refactorer) [10], and EUnit for testing Wrangler.

In our experience, the main disadvantage of writing test cases under this approach is that it is very tedious, and hard to cover all the refactoring scenarios especially when both the implementation and the test cases are written by the same people. Hence we did not gain sufficient assurance about the correctness of the refactorings implemented.

Other variants of this approach involve a degree of random generation; we will explore our particular approach in the next section, and discuss related work in section 8.

4.4 Program Verification

Rather than using testing, it is possible to write formal proofs of correctness for refactoring engines. Two approaches suggest themselves:

- It is possible to produce, program by program, separate proofs of equivalence between the original and the refactored programs. Such proofs might be generated by tactic-based proof descriptions, or result from a proof planning process.
- Alternatively, the formal theorem proved can itself contain a quantifier over all programs of a certain form (which are the input to the refactoring in question). Preliminary work under this approach is to be found in Li's thesis [9] and the forthcoming thesis of Sultana [15].

This section has summarised various approaches to validating refactoring engines; we next look at our particular work.

5 Testing Wrangler with QuickCheck

Before adopting QuickCheck as the test engine of refactorings, we used the unit testing approach, as discussed in the previous section. We concluded that this mechanism was not ideal, and so to improve the testing of Wrangler, we have experimented with the idea of using Quviq QuickCheck as the test engine.

Under this approach, a collection of properties are written along with the implementation of each refactoring. These properties specify the conditions that must be met by the program after the refactoring, in order for the transformation to be behaviour-preserving. From the formal specification point of view, these properties can be viewed as the post-conditions of a refactoring. While there are some general properties which apply to most of the refactorings, for example, all the programs after a refactoring must compile successfully, some properties are particular to individual refactorings, especially those involving structural changes to the program. Writing properties along with the implementation of refactorings, we are able to make testing an integral part of the refactoring development process.

Properties are tested on the refactored version of the input program. While occasionally we have written a few small input programs to test a particular case, mostly we use real-world Erlang programs as the testing code base. Before

the testing of a specific refactoring, the code base could be examined to make sure that enough refactoring scenarios are covered in the program. For example, to test a refactoring involving the communication between processes, we should choose programs that contain substantial process communications; and to test a refactoring that transforms a tuple to a record, we need to make sure that tuples and records are reasonably used in the test program. Apart from manual examination, the `collect/1` function provided by Quviq QuickCheck can be used to analyse the distribution of the test data when the testing is complete; and the coverage analysis functionalities provided by the standard Erlang release can be used to analyse how well the code implementing the refactoring is covered by running the test cases.

Once the test program has been chosen, refactoring commands are automatically generated using the information stored in the annotated abstract syntax tree (AAST) of the test program. Both the generation of refactoring commands and the creation of properties make use of the Wrangler infrastructure API. The API provides programmer access to the infrastructure on which Wrangler is built. As the infrastructure has been more thoroughly tested, we trust its robustness in this exercise. Alternatively, we can also test an API function exposed by the infrastructure using the same approach.

More about the generation of refactoring commands and the creation of properties are discussed in the following two sub-sections. Following that, as an example, testing of the *renaming a function* refactoring is examined in more detail.

5.1 Generation of Refactoring Commands

In Wrangler, a refactoring command normally contains the refactoring name, the name of the source file under refactoring, the focus of the refactoring which can be a location/range in the program source, and some user inputs. For example, the refactoring *renaming a function* has the following interface:

```
rename_fun(FileName, SrcLoc={Line, Col}, NewName, SearchPaths)
```

where `FileName` is the name of the file containing the definition of the function to be renamed; `SrcLoc`, which is a tuple containing a line and a column number, represents the location of one of the occurrences of the function name in the source; `NewName` is the new function name, and `SearchPaths` specifies where to search for those files that could possibly use this function; this is needed when the function to be renamed is exported by the module in which it is defined.

As another example, the refactoring *generalisation of a function definition* has the following interface:

```
generalise_fun(FileName, Range={StartLoc, EndLoc}, ParName)
```

where `FileName` is the name of the source file containing the definition of the function to be generalised; `Range` represents the start and end location of the selected expression in the source, and `ParName` is the new parameter name. As this refactoring only affects the current module, `SearchPaths` is not needed.

Next, we return to the 'renaming' example to explain how refactoring commands can be generated. If a specific file is used as the input program, then the `FileName` is fixed, otherwise a file can be randomly chosen from a directory for each refactoring command. The following function serves to select an Erlang file from a directory.

```
gen_filename(Dir) ->
    {ok,Files} = file:list_dir(Dir),
    ErlFiles = [F|| F <-Files, filename:extension(F)==".erl"],
    oneof(ErlFiles).
```

where the function `oneof/1` is a QuickCheck API function which generates a value using a randomly chosen element of a list of generators; in this example, all the list elements are constant generators.

Instead of generating source locations using the integer generators provided by Quviq QuickCheck, the value of `SrcLoc` is generated based on the location information stored in the AAST representation of the chosen Erlang file. As discussed earlier, in the AAST, each occurrence of a function name is associated with its location in the source, the name of the module in which it is defined, as well as its defining location in that module.

To generate a source location, we first collect all those locations which are associated with the occurrences of function names defined in this file, then choose one from the collection randomly. This way, we can make sure that selected location points to a function name defined in the current module. In order to test the case when the user deliberately points to a location in the source which does not correspond to a function name defined in the module, we can always add fake locations to the collection of real ones, or make use of QuickCheck's fault injection combinators: `fault/1` and `fault_rate/3`.

Some refactorings ask the user to input a new name. For example, to rename a function, the user needs to input the new function name; and to generalise a function definition, the user has to input a new variable name. To improve the possibility that a name conflict/shadow occurs, identifier names are generated from both pre-created fresh names and those used in the refactored program, since a name conflict/shadow is possible only when the new name is already used by program.

The following function generates refactoring commands for *renaming a function*.

```
rename_fun_commands(Dir) ->
        ?LET(FileName, gen_filename(Dir),
        {FileName,
         oneof(collect_fun_locs(FileName)),
         oneof(collect_names(FileName)),
         Dir}).
```

In the above function, `Dir` specifies where to look for Erlang files to refactor; `?LET` is a macro provided by Quviq QuickCheck (`?LET(Pat, G1, G2)` generates

a value from `G1`, binds it to `Pat`, then generates a value from `G2` which may refer to the variables bound in `Pat`); function `collect_fun_locs/1` adds all the locations where a locally defined function name occurs in the selected Erlang file to a list of default locations; `collect_name/1` adds all the function names that occur in the source to a list of pre-created fresh identifiers, and as last, we assume that `Dir` is the only directory to search for those files that would possibly be affected by the refactoring.

Suppose that the testing directory is `"c:/wrangler-0.1/test"`, which has three Erlang files, the following shows part of the refactoring commands generated by the above function in one run of QuickCheck.

```
1% {"test.erl",{3,1},module,"c:/wrangler-0.1/test"}
1% {"refac_rename_fun.erl",{243,64},halt,"c:/wrangler-0.1/test"}
1% {"refac_qc.erl",{184,48},ordsets,"c:/wrangler-0.1/test"}
1% {"test.erl",{5,39},"DDD","c:/wrangler-0.1/test"}
1% {"refac_qc.erl",{366,30},get_pos,"c:/wrangler-0.1/test"}
1% {"refac_rename_fun.erl",{117,33},purge_module,"c:/wrangler-0.1/test"}
```

As an example, the first command means to rename the function whose name occurs at the location: {`line: 3, column: 1`} in file `test.erl` to the new name `module`, and search the directory `"c:/wrangler-0.01/test"` for files in which the function is used, if the function is exported. The percentage at the beginning of each line shows the proportion of the total represented by the command.

5.2 Properties

Formally specified or not, each refactoring comes with a set of pre-conditions, which embody when a refactoring can be applied to a program without changing its meaning; a set of transformation rules which state how the program should be transformed to fulfil the refactoring while keeping the program's semantics unchanged; and a collection of post-conditions which articulate some properties the program should hold after the refactoring has been done. While the pre-condition checking and transformation rules are always explicitly implemented, the checking of post-conditions are normally ignored by the developers of refactoring tools as we assume that the pre-conditions and transformation rules together should guarantee the post-conditions.

With the QuickCheck testing approach, we can test most of these post-conditions explicitly. Ideally, one post-condition that applies to any refactoring is that the input program and its refactored version should have the same semantics; however whether two programs have the same semantics is in general not decidable. Furthermore, even when the two programs have the same semantics, the refactor still might not have performed the anticipated structural change to the program correctly as mentioned before. Therefore, instead of checking two programs having the same semantics, we test a number of properties that are decidable.

There are a couple of basic properties that should hold by all the refactorings:

- first, the refactoring engine should not crash, i.e. the refactorer should not terminate with an uncaught exception;
- second, if the refactoring has finished without giving an error message, then the refactored version of the program should compile successfully (Wrangler only refactors programs that compile).

Many basic refactorings are bi-directional. Given a refactoring that transforms a program from P to P', we can generally find another refactoring that transforms program P' to P. For example, renaming an entity in a program from A to B, then renaming it back to A, should produce the original program; as another example, first generalising a function definition over an expression, then specialising the function on the newly added parameter with the expression should always produce the original function. This feature of refactoring allows us to write properties that embody mutual testing of refactorings.

During the implementation of Wrangler, we always try to separate the precondition checking part from the transformation part. One of the benefits of doing this is that it allows the mutual testing of condition-checking and transformation. For example, performing the transformation with the knowledge that some of the necessary side-conditions are not satisfied should either make the refactoring engine crash or violate some post-conditions in the case that the transformation (apparently) succeeds.

Apart from those general post-conditions that apply to most of refactorings, each refactoring also has it own particular post-conditions, especially those concerning structural changes of the program, as different refactorings change the program structure in different ways. For some refactorings, there may also be special constraints that should hold during the transformation. For example, some refactorings are suppose to keep the program's module interface unchanged; while others are suppose to keep some particular function interfaces unchanged. All these constraints can be expressed as QuickCheck properties.

There is no limit on the number of properties one can specify to test a refactoring. For complex transformations, instead of writing a small number of very complex properties, we can always write a collection of simpler properties, each of which specifies only one aspect or a small step of the transformation. Simpler properties are easier to understand, maintain and reuse.

In the following section, we again take the *renaming a function* refactoring as an example to illustrate how properties can be specified and tested.

6 An Example: Testing *Renaming a Function*

Renaming a function is one of the most basic, but very useful, refactorings, supported by almost all the existing refactorers. This refactoring renames a user-selected function name to a new name and updates all the references to it. When the renamed function is exported by the module, this refactoring could potentially affect every module in the program. Suppose the old and new function

names (with arity) are `bar/n` and `foo/n` respectively, then the side-conditions on *renaming a function* are as follows.

1. The new name should be a lexically valid function name, otherwise the transformed program will not compile.
2. No binding for `foo/n` may exist in the same scope. This condition avoids *name conflict* in the scope where `bar/n` is defined, and violating this condition will result in the transformed program failing to compile.
3. No binding for `foo/n` may intervene between the binding of `bar/n` and any of its uses, and the binding to be renamed must not intervene between existing bindings and the uses of `foo/n`.

   This condition avoids *name capture*, and violating this condition will lead to the binding structure of the program being changed silently. ('Binding structure' here refers to the association of uses of identifiers with their definitions in a program, and is determined by the scope of the identifiers).
4. Callback functions should not be renamed. Callback functions in Erlang are generally named by Erlang OTP behaviours, and must be implemented by the module calling an OTP behaviour. Renaming, or changing the interface of, callback functions in either single side will break the protocol between the OTP behaviour and the module calling the OTP behaviour, and make the program fail to function properly.

To check the correctness of the implementation, we focus on defining properties depending on whether the refactoring succeeds or not. If the refactoring completes without giving an error message, we then test the following properties.

- Renaming the new function back to its original name should affect the same set of Erlang files in the application, and produce the original program except for variations of layout. This property also implies the condition that the refactored version of the program should compile without errors.
- The function-level binding structure of the refactored version of the program should be the same as, or isomorphic to, that of the original program.

  Unlike some functional languages that allow nested function definitions, Erlang has a very straightforward function defining structure. In Erlang, all named functions are top-level functions. The function-level binding structure of an Erlang program can be represented as a list of tuples:

$$B = [\{\{M_1, Loc\}, \{M_2, Id, A\}\}]$$

  and $\{\{M_1, Loc\}, \{M_2, Id, A\}\} \in B$ if and only if the function name Id, which occurs in module M_1 at location Loc, refers to the function defined in M_2 whose name is Id and arity is A. To take the possible program layout change into account, Loc here is a number reflecting the function name's textual occurrence order in the code, instead of the concrete source location.

  Suppose the function `bar/1` defined in module `N` is renamed to `foo/1`, and the binding structures of the program before and after the refactoring are B and B' respectively, then replacing all the occurrences of $\{N, \texttt{foo}, 1\}$ in B' with $\{N, \texttt{bar}, 1\}$ should produce B.

```
qc_rename_fun(Dir) ->
  F = ?FORALL(C, (rename_fun_commands(Dir)),
  begin
    [FileName, SrcLoc, NewName, SearchPaths] = C,
      %% backup the current version of the program.
    file:copy(FileName, "temp.erl"),
      %% get the function name (with arity) to be renamed.
    {Mod, FunName, Arity} = pos_to_fun_name(FileName, SrcLoc),
      %% calculate the binding structure of the current program.
    B1 = fun_binding_structure(FileName),
      %% get the name of the callbacks functions if there is any.
    CallBacks = get_callback_funs(FileName),
      %% apply the refactoring command to the source.
    Res = apply(refac_rename_fun, rename_fun, C),
    case Res of
        %% ChangeFiles contains the names of those files
        %% that have been affected by this refactoring.
      {ok, ChangedFiles} ->  %% refactoring completed successfully.
        B2 = fun_binding_structure(FileName), %% new binding structure.
          %% get the name of the callback functions if there is any.
        CallBacks1 = get_callback_funs(FileName),
        C1 = [FileName,NewName, Arity, FunName, SearchPaths],
          %% rename the function back to its original name.
          %% we cannot use location as it might have been changed.
        {ok, ChangedFiles1} = apply(refac_rename_fun, rename_fun_1, C1),
          %% property1: renaming in both directions affect the same set of files.
        prop1 = ChangedFiles == ChangedFiles1,
          %% property2: rename twice should returns to the original file.
        Prop2 = pretty_print(FileName) == pretty_print("temp.erl"),
          %% property 3: B1 and B2 are isomorphic.
          %% rename/3 replaces Mod, FunName, Arity with Mod, NewName, Arity in B1
        Prop3 = B2== rename(B1, {Mod, FunName, Arity}, {Mod, NewName, Arity}),
          %% property 4: the same set of callback functions.
        Prop4 = CallBacks == CallBacks1,
          %% recover the original program for the next refactoring command.
        file:copy("temp.erl", FileName),
        Prop1 and Prop2 and Prop3 and Prop4;
      {error,ErrorMsg} ->  %% refactoring failed with an error message.
        %% carry out the transformation even though the side-conditions
        %% do not held; do_rename_fun/4 transforms the program.
      _Res = apply(refac_rename_fun, do_rename_fun, C),
      case ErrorMsg of
        {1, _R1} ->     %% failed for side-condition 1;
                      %% the transformed program should not compile.
                    file:copy("temp.erl", FileName),
                    {error, _Reason} = get_AST(FileName), true;
        {2, _R2} ->     %% failed for side-condition 2;
                    file:copy("temp.erl", FileName),
                    {error, _Reason} = get_AST(FileName), true;
        {3, _R3} ->     %% failed for side-condition 3;
                      %% the transformed program should compile, but the new
                      %% binding structure is not isomorphic to the original one.
                    {ok, _AST} = get_AST(FileName),
                    B2 = fun_binding_structure(FileName),
                    file:copy("temp.erl", FileName),
                    B2 /= rename(B1, {Mod,FunName, Arity},{Mod, NewName, Arity})
      end end end),
    qc:quickcheck(F).
```

Fig. 3. The top-level function for testing *renaming a function*

This property is able to find certain bugs that escape detection by the previous property. For example, an implementation that renames every occurrence of the selected function name irrespective of its semantics will be found faulty by this property, but not necessarily by the previous property.

– The programs before and after the refactoring should have the same set of callback functions if which functions are callback functions has been explicitly specified.

If the refactoring fails because one of the side-conditions fails, then the necessity of the side-condition can also be tested. For example

– Transforming the program when side-condition 1 or 2 does not hold should produce a program that does not compile.
– Transforming the program when side-condition 3 does not hold should produce a program that compiles but has a different function-level binding structure.

A simplified version of the top-level function for testing *renaming a function* is given in figure 3. To make it easier to read, we have omitted the part that handles client modules, however this should not affect the idea expressed by this function.

7 Evaluation of Approach

A number of other refactorings have been tested using this approach, including *renaming a variable name*, *generalisation of a function definition*, etc. We actually started to use Quviq QuickCheck after the first preliminary release of Wrangler, which was tested on a number of small test cases using EUnit, and was also manually tested on a large code base.

Even so four bugs were found within the first release of Wrangler in a short time. All these bugs escaped the pre-release testing due to the incomplete coverage of the testing suite. Among these bugs, one silently changed the binding structure of the program when the *generalisation* refactoring is applied, and was detected by a property we wrote for this refactoring, which states that *generalisation* and *specialisation* are inverse; the other three bugs were all caught by the very basic properties, for example, one bug caused the refactoring engine to crash because of an unmatched case clause; and another caused the refactored code fail to compile because of the improper handling of generalisation on operators.

From our experience so far, the advantages of the QuickCheck approach are as follows:

– We are able to make the development of refactorings and their testing very closely integrated. The meaning of each refactoring was further clarified by the mutual testing of the implementation and the specification.
– Once properties have been written, many different test cases can be run with very little effort, instead of repeating the same set of test cases every time. As any Erlang program can serve as the test program, we can run the testing on as many test programs, especially large programs, as possible.
– Because of the controlled random generation of refactoring commands, and the large amount of tests we can run, more refactoring scenarios will be

covered, therefore increasing the possibility of finding more bugs. At this point, one might think of the exhaustive testing of refactorings. While it is possible to enumerate all the possible refactoring commands when the input program is very small, it is not practical with large input programs due to the huge amount of refactoring commands that could be generated.
- This approach scales well to complex refactorings or composite refactorings. Testing of a complex refactoring does not necessitate the specification of very complex properties. Instead, we could write a collection of simple properties, each of which only tests one aspect of the refactoring. A composite refactorings can usually be decomposed into a series of basic refactorings, and each of these basic refactorings can be tested separately using this approach. This also corresponds naturally to the implementation of composite refactorings

While properties can be written separately from the implementation of refactorings, these properties normally make use of the infrastructure on which the refactorings are built, therefore familiarity with the infrastructure is essential for the testing using this approach.

8 Related Work

A number of case studies regarding to the use of Quviq QuickCheck or its predecessor as the test engine have been done and reported, among which one to test an industrial implementation of the Megaco protocol, and faults that have not been detected by other testing techniques were found [2]. This case study also shows the power of shrinking provided by Quviq QuickCheck, and one example is that a test case consisting of a sequence of 160 commands was reduced to just seven. Shrinking of refactoring commands does not make the counterexample any simpler, therefore plays little role in this case study.

The most closely related work on the automated testing of refactorings is the approach of Daniel *et. al.* [5]. The core of this approach is ASTGen, a library for generating abstract syntax trees (ASTs) for Java programs. ASTGen allows the developer to write *imperative generators* whose executions produce abstract syntax trees (ASTs) for refactoring engines. To test a refactoring, a developer writes a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. Several kinds of properties (oracles) have also been created to automatically check that the refactoring engine transformed the generated program correctly. Compared with this approach, our approach is more lightweight, however a developer does need to make sure that the testing code base covers enough structure features and refactoring scenarios for the refactoring under testing.

9 Conclusion

Refactoring tools ought to allow program developers to quickly and safely refactor their program, especially large programs. However, a robust and safe refactoring tool is hard to develop, and most refactoring tools still contain bugs even

after extensive testing. While unit testing does help to find bugs in refactoring tools, it is tedious to manually write test programs, and the coverage of the test cases is hard to guarantee, and it is even harder to test refactoring tools on large systems.

We have explored the idea of using Quviq QuickCheck to automate the testing of refactorings. In this approach, the correctness of refactorings is tested against specifications written in Erlang. Once a test program has been chosen, we automated the generation of refactoring commands and the checking of refactoring outputs. Within a short time, a number of bugs were found in the first release of Wrangler using this approach. The pros and cons of this approach is summarised in section 7.

We envisage exploring a number of further ideas for automated testing of refactorings using QuickCheck.

- It would be also interesting to generate Erlang programs to be refactored to see whether more combinations of Erlang constructions that provoke faults in Wrangler can be found.
- One of the options followed by Daniel *et. al.* in [5] is to compare the effect of two refactoring engines, namely Eclipse and NetBeans for Java. We will explore this option for Wrangler and the refactoring engine built by the group at Eötvös Loránd University, Budapest [12].
- We have not addressed the behaviour checking of programs; it would nevertheless be possible to extend our work to check the results of refactorings against their original version using randomly-generated input values.
- We have assumed the correctness of our infrastructure library; it would be instructive to express and then to test crucial properties of the functions in this library.

We also intend to provide an API to help the specification of properties in the context of refactorings, and we would also like to adopt this approach to test our Haskell refactoring tool, HaRe.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. 2nd edn. Prentice-Hall, Englewood Cliffs (1996)
2. Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing Telecoms Software with Quviq QuickCheck. In: Trinder, P. (ed.) Proceedings of the Fifth ACM SIGPLAN Erlang Workshop. ACM Press, New York (2006)
3. Carlsson, R., Rémond, M.: Eunit: a lightweight unit testing framework for erlang. In: ERLANG 2006: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang, p. 1. ACM Press, New York (2006)
4. Claessen, K., Hughes, J.: QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. ACM SIGPLAN Notices 35(9), 268–279 (2000)
5. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: ESEC/FSE 2007: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 2007. ACM Press, New York (2007)

6. FORSE. Formally-Based Tool Support for Erlang Development, http://www.cs.kent.ac.uk/projects/forse/
7. Fowler, M.: Refactoring Home Page, http://www.refactoring.com
8. Hughes, J.: QuickCheck Testing for Fun and Profit. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 1–32. Springer, Heidelberg (2006)
9. Li, H.: Refactoring Haskell Programs. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK (September 2006)
10. Li, H., Reinke, C., Thompson, S.: Tool Support for Refactoring Functional Programs. In: Jeuring, J. (ed.) ACM SIGPLAN Haskell Workshop, Uppsala, Sweden (August 2003)
11. Li, H., Thompson, S.: A Comparative Study of Refactoring Haskell and Erlang Programs. In: Di Penta, M., Moonen, L. (eds.) Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), September 2006, pp. 197–206. IEEE, Los Alamitos (2006)
12. Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring Erlang Programs. In: The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden (November 2006)
13. Gorrie, L.: Distel: Distributed Emacs Lisp (for Erlang). In: Eighth International Erlang/OTP User Conference
14. Carlsson, R.: Erlang Syntax Tools, http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3
15. Sultana, N.: Verification of Refactorings in Isabelle/HOL. Master's thesis, Computing Laboratory, University of Kent, UK (September 2007)

Optimal Lambda Lifting in Quadratic Time

Marco T. Morazán and Ulrik P. Schultz

Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu
University of Southern Denmark, Odense, Denmark
ups@mmmi.sdu.dk

Abstract. The process of lambda lifting flattens a program by lifting all local function definitions to the global level. Optimal lambda lifting computes the minimal set of extraneous parameters needed by each function as is done by the $O(n^3)$ equation-based algorithm proposed by Johnsson. In contrast, modern lambda lifting algorithms have used a graph-based approach to compute the set of extraneous parameters needed by each function. Danvy and Schultz proposed an algorithm that reduced the complexity of lambda lifting from $O(n^3)$ to $O(n^2)$. Their algorithm, however, is an approximation of optimal lambda lifting. Morazán and Mucha proposed an optimal graph-based algorithm at the expense of raising the complexity to $O(n^3)$. Their algorithm, however, suggested that dominator trees might be used to develop an $O(n^2)$ algorithm. This article explores the relationship between the call graph of a program, its dominator tree, and lambda lifting by developing algorithms for successively richer sets of programs. The result of this exploration is an $O(n^2)$ optimal lambda lifting algorithm.

1 Introduction

The process of lambda lifting flattens a program by lifting all local function definitions to the global level. In order to perform this program transformation the free variables of a function, f, and a subset of the free variables transitively needed by its callees, must be added as formal parameters to f before it can be lifted to the global level. That is, f must be made scope insensitive before it can be moved to the global level. Free variables must be explicitly passed to f, because at runtime the lifted version of f does not have the benefit of a closure to store the bindings of the free variables. This program transformation technique is valid for programs using higher-order functions, because the extraneous parameters are passed to function references (e.g. the site where functions are passed as arguments) rather than to function calls (e.g. the site where a function is applied to a set of arguments).

Lambda lifting is important for restructuring functional programs written for the web [7], for partial evaluators [1], and for efficient compilation [15]. Furthermore, many abstract machines for functional languages only handle lambda-lifted programs [10,12] making this transformation an important step in several compilers for functional languages [9,11]. Lambda lifting and its inverse lambda

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 37–56, 2008.

dropping [2] are also important for improving the performance of compiled programs by providing a mechanism through which the number of parameters of a function can be optimized for the target machine. For example, functions with a large number of parameters (which are handled poorly by most compilers) can be transformed to have fewer parameters [2]. Danvy and Schultz also point out that in the context of teaching, lambda lifting and lambda dropping are useful by offering different views of programs that help students understand lexical scoping and block structure [2].

The computation of the set of free variables needed by a lifted function makes lambda lifting difficult. Modern graph-based approaches [3,14] tackle the problem by transforming the call graph of a program into a directed acyclic graph that is used to propagate free variables. The algorithm developed by Danvy and Schultz [3] improves the complexity of Johnsson's [8] lambda lifting algorithm from $O(n^3)$ to $O(n^2)$. Their algorithm, however, is not optimal because it may unnecessarily increase the arity of lifted functions. The algorithm developed by Morazán and Mucha [14] makes graph-based lambda lifting optimal at the cost of increasing its complexity to $O(n^3)$.

In this article, we first review Johnsson's (J), Danvy's and Schultz's (DS), and Morazán's and Mucha's (MM) lambda lifting algorithms. After this review, we present a new insight that simplifies the presentation and the implementation of graph-based lambda lifting by using a depth-first traversal instead of a reversed breadth-first traversal to propagate free variables. The article then explores the relationship between call graphs, dominator trees, and lambda lifting. The result of this exploration is an optimal $O(n^2)$ lambda lifting algorithm. Although the discussion is technically intricate at some points, the resulting algorithm is simple and elegant. The presentation assumes that all variable names are unique. Programs for which this does not hold can easily be transformed by generating a fresh identifier for repeated identifiers [4]. Moreover, since lambda-lifting (as pointed out earlier) is indifferent to higher-order functions, our presentation only uses first-order programs as examples. The article ends with some concluding remarks and directions of future work. The appendix includes a brief glossary of the graph terminology used in the article (i.e. tree, dominator tree, and strongly connected component).

2 Lambda Lifting Algorithms

2.1 Johnsson's Algorithm

In the J-algorithm, the source program is traversed top-down to compute the required (i.e. minimal) set of extraneous parameters needed by each function. For any given function, f, the equation for the required set of free variables of f, R_f, is given by:

$$R_f = FV_f \cup ((\cup_{g \in FF_f} R_g) \cap SV_f), \tag{1}$$

where FV_f is the set of free variables directly referenced by f, FF_f is the set of functions referenced by f, and SV_f is the set of variables defined in f's enclosing

```
(define (f x y)
  (define (g...) (...x...i...))
  (define (h...) (...y...f...))
  (define (i...) (...f...))
 (...g...h...))
```

Fig. 1. First Scheme Pseudo-code

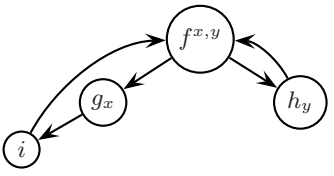


Fig. 2. Call Graph

lexical scope. Mutually recursive functions give rise to a system of mutually recursive equations which is solved by traversing down the parse tree. Once R_f is known it is used to compute the minimal set of free variables for functions declared further down the program's parse tree.

To illustrate how the J-algorithm works using equation (1) consider the pseudo-code in Figure 1. At the topmost level of the parse tree the free variables of f are computed by solving the following equation:

$$R_f = FV_f \cup \ ((\cup_{g \epsilon FF_f} \ R_g) \ \bigcap \ SV_f) \ .$$

Since $FV_f = SV_f = \emptyset$, we may conclude that $R_f = \emptyset$.

At the next level of the parse tree, the free variables equations to solve are:

$$\begin{aligned}
R_g &= FV_g \cup \ ((\cup_{j \epsilon FF_g} R_j) \cap SV_g) \\
&= \{x\} \cup \{R_i \cap \{x, y\}\} \\
&= \{x\} \cup \{\{FV_i \ \cup ((\cup_{j \epsilon FF_i} R_j) \cap SV_i)\} \cap \{x, y\}\} \\
&= \{x\} \cup \{\{\emptyset \cup \{R_f \cap \{x, y\}\}\} \cap \ \{x, y\}\} \\
&= \{x\} \cup \ \{\emptyset \cup \{\emptyset \cap \{x, y\}\} \cap \ \{x, y\}\} \\
&= \{x\} \\
R_h &= FV_h \cup ((\cup_{j \epsilon FF_h} R_j) \cap SV_H) \\
&= \{y\} \cup \{R_f \cap \{x, y\}\} \\
&= \{y\} \cup \{\emptyset \cap \{x, y\}\} \\
&= \{y\} \cup \emptyset \\
&= \{y\} \\
R_i &= FV_i \cup ((\cup_{j \epsilon FF_i} \ R_j) \cap SV_i) \\
&= \emptyset \cup \{FV_f \cap \{x, y\}\} \\
&= \emptyset \cup \{\emptyset \cap \{x, y\} \\
&= \emptyset \cup \emptyset \\
&= \emptyset
\end{aligned}$$

Notice that x is not identified as an extraneous parameter needed by h and that y is not identified as an extraneous parameter needed by g nor i. Furthermore, x is not identified as an extraneous parameter for i. This occurs, because the set of extraneous parameters needed by f, an ancestor of g, h, and i in the program's parse tree, are computed before the set of extraneous parameters needed by g, h, and i. Thus, the members of FF_f are not explored during the computation of R_g, R_h, and R_i and do not contribute extraneous parameters to g, h, and i.

The time complexity of the J-algorithm is $O(n^3)$, where n is the size of the program. Briefly, globally there are $O(n)$ equations to solve the transitive closure problem, which requires $O(n)$ steps of set union operations each taking $O(n)$ per equation.

2.2 Danvy's and Schultz's Graph-Based Lambda Lifting

To perform lambda lifting in quadratic time, a program is represented as a call graph. Each node in this graph represents a function. An edge from f to g means that there is a reference to g in the body of f. Mutually recursive functions give rise to strongly connected components (akin to Johnsson's mutually recursive equations). Danvy and Schultz observed that a function, f, in a strongly connected component can be given as extraneous parameters the set of free variables lexically visible to f found in the union of the free variables of the functions that constitute the component. Therefore, strongly connected components can be coalesced in the call graph of a program to yield a directed acyclic graph that is traversed in a reversed breadth-first order to propagate free variables between nodes.

To illustrate the DS-algorithm consider the call graph in Figure 2 for the pseudo-code in Figure 1. In the call graph each node is labeled with the name of a function. The superscript at the right of each function name is the set of variables declared by the function that appear in the pseudo-code and the subscript at the right of each function name is the set of free variables referenced by the function. The nodes in the call graph form a strongly connected component and are coalesced yielding a graph with a single node. The union of all the free variables of the functions in the node (i.e. f, g, h, and i) is taken. For each function, the lexically visible variables in this union become parameters to the lifted functions. That is, $\{x, y\}$ are identified as extraneous parameters for g, h, and i.

The time complexity of the DS-algorithm is $O(n^2)$, where n is the size of the program. Briefly, the coalesced call graph of size $O(n)$ defines a global order in which free variables can be linearly propagated through the graph, with each propagation step performing a set unification of size $O(n)$.

2.3 Morazán's and Mucha's Graph-Based Algorithm

Morazán and Mucha observed that using strongly connected components to propagate free variables may result in an approximation of the required set of extraneous parameters needed by lifted functions as exemplified by the results obtained by the J-algorithm and the DS-algorithm for the pseudo-code in Figure 1. Unnecessary extraneous parameters may be added to lifted functions for two reasons. The first reason is that functions can be members of a strongly connected component that contains nested strongly connected components and that also contains functions defined at different levels in the program's parse tree. Suppose r is a function defined at level n in the parse tree of a program and that there are m disjoint sets of functions (modulo r), $D_1 \ldots D_m$, defined

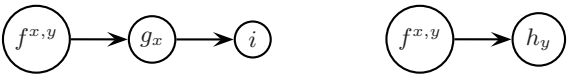


Fig. 3. MM-Algorithm Components for the Call Graph in Figure 2

at any level greater than n (i.e. in the parse tree of the program r is an ancestor of these functions) such that r dominates all paths from functions in D_i to functions in D_j, $i \neq j$. In such a scenario, r may declare variables that are free[1] for functions in D_i that are not needed as extraneous parameters by functions in D_j and viceversa. This may occur, for example, when r is contained in two independent loops (modulo r).

The second reason is that a variable, x, declared by r that is free in D_i may not be needed as an extraneous parameter by all the functions in D_i. For example, let r and s be members of the same loop such that x is known to be free in s and is declared by r. The variable x only needs to be carried by successors of s in the call graph if there is a path, that does not contain r, from s to another function where x is directly referenced. This follows from the observation that the successors of s do not need to make x available to any other function if such a path does not exist. Thus, these successors do not require x as an extraneous parameter.

The MM-algorithm is an improvement of the DS-algorithm that reduces the arity of lifted functions by computing the minimal set of extraneous parameters needed by each lifted function, as is done by the J-algorithm, based on the observations above. Extraneous parameters are reduced by splitting the strongly connected components of a call graph that contain functions defined at different levels in the program's parse tree into multiple components based on its nested strongly connected components and by ignoring edges into a dominating function that are internal to any such component after the split. Splitting strongly connected components into multiple components guarantees that free variables local to a component (e.g. declared by the dominating function) do not propagate between nested strongly connected components. Ignoring internal edges into the dominating function of a nested strongly connected component guarantees that a free variable local to a component is not propagated beyond the last function that references it in a loop. This occurs, because the removal of such edges eliminates the loop and, therefore, these functions no longer constitute a strongly connected component and do not receive the same set of extraneous parameters.

To illustrate the MM-algorithm once again consider the call graph in Figure 2. The graph is split into two components displayed in Figure 3. This disconnected graph is used to propagate free variables between nodes in a reversed breath-first order. Notice that the dominating ancestor function, f, is a member of two components which prevents its descendants from unnecessarily contributing free variables to each other. By ignoring the edges into f in Figure 2, the nested

[1] We call such free variables *local* to the strongly connected component.

```
(define (f x)
  (define (g...) (...i...))
  (define (h...) (...x...))
  (define (i...) (...h...))
 (...g...h...))
```

Fig. 4. Second Scheme Pseudo-code

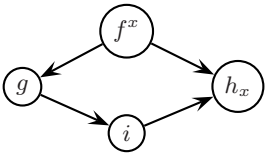


Fig. 5. Call Graph for Figure 4

strongly connected components cease themselves to be strongly connected. During the propagation of free variables, y is not unnecessarily propagated to g and i, and x is not unnecessarily propagated to h and i. Notice that within the loop formed by $\{f, g, i\}$ in Figure 2 only g requires and receives x as an extraneous parameter. This algorithm yields the same results as the J-algorithm.

The time complexity of the MM-algorithm is $O(n^3)$, where n is the size of the program. Briefly, strongly connected components must be split $O(n)$ times. For each split the resulting coalesced call graph is of size $O(n)$ and each propagation step performs a set unification of size $O(n)$.

3 A Simplifying Insight

The graph-based lambda lifting algorithms developed to date use the reversed breadth-first ordering of the nodes of an acyclic graph to ensure that a node is only processed once all of its successors in the call graph have been processed. Successor nodes must be processed first, because the required set of free variables of predecessor nodes depends on them. The use of this ordering, however, requires that special attention be paid to calls from functions appearing late in the reversed breadth-first ordering to functions appearing early in the reversed breadth-first ordering.

To illustrate the problem consider the Scheme pseudo-code in Figure 4 and its diamond-shaped call graph in Figure 5. In this graph the function f declares x (noted as right superscript) and x is free in h (noted as a right subscript). The breadth-first ordering of the nodes is: $\{f, g, h, i\}$[2]. There are no strongly connected components and, thus, nothing to coalesce. Having an acyclic graph means that free variables ought to be propagated from callees to callers in a reversed breadth-first order. For our example that order is: $\{i, h, g, f\}$. If free variables are simply propagated from callees to callers nothing propagates from i to g, from h the free variable x propagates to i and nothing propagates to f, and nothing propagates from g to f. The end result is that x is identified as a free variable for h and i, but not for g which also needs x as a free variable. To avoid this pitfall, the DS-algorithm unifies the set of local free variables with the set of free variables of the immediate successors in the call graph. Thus, x propagates from i to g when g is processed.

[2] The breadth-first ordering could also be $\{f, h, g, i\}$, but this is irrelevant for our purposes.

We observe that if the graph is acyclic, as is always the case after coalescing strongly connected components, then a depth-first traversal of the graph can be used to propagate free variables: every time the process pops back from a node to its antecessor free variables are propagated. This ensures that all successors are processed before a caller is processed. For the call graph in Figure 5, a depth-first traversal follows the path $f \rightarrow g \rightarrow i \rightarrow h$. The free variable x is propagated back through this path from h to i and finally to g. The depth-first traversal then proceeds down the path $f \rightarrow h$ and nothing additional is propagated from h to f before terminating. Although the result is the same as using the reversed breadth-first ordering, this process is more elegant and simplifies the implementation of lambda lifting.

Propagation using a depth-first traversal instead of a reversed breadth-first ordering is still proportional to the number of function calls and the number of declared variables in the program. This type of traversal does not change the time complexity of neither the DS-algorithm nor the MM-algorithm.

4 Call Graphs and Dominator Trees

The key lessons that must be highlighted from the previous sections are:

1. **J-algorithm:** The set of extraneous parameters for an ancestor function in a parse tree must be known before finalizing the computation of the set of extraneous parameters for any descendant of this function.
2. **DS-algorithm:** Lambda lifting can be done using a graph-based approach. Furthermore, functions in a strongly connected component of a call graph that do not have references to any free variables local to the component can be coalesced. These functions all require the same set of extraneous parameters.
3. **MM-algorithm:** Dominating functions must not be coalesced with their dominated functions in order to avoid dominated functions from unnecessarily contributing free variables to each other. Furthermore, simple loops on a dominating function must be dissolved in order to avoid unnecessary propagation of free variables.
4. **New Observation:** Once an acyclic graph is obtained for a graph-based approach a depth-first traversal can be used to simplify the process of propagating free variables.

The MM-algorithm repeatedly computes strongly connected components in order to avoid the unnecessary propagation of local free variables. The splitting of a strongly connected component is always done around an ancestor function that dominates all paths between disjoint sets of functions within the strongly connected component when the strongly connected component contains functions defined at different levels in the program's parse tree. This observation suggests that dominator trees can be used to perform lambda lifting.

A defining property of a dominator tree is that an ancestor function always appears before its descendants. Thus, a dominator tree tells us for which functions the complete set of free variables must be computed first. For our purposes,

```
(define (f x y z)
  (define (g a b)
    (define (h c d)
      (define (i e) (...h...g...))
      (...i...))
    (...h...))
  (...g...))
```

Fig. 6. Third Scheme Pseudo-code

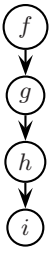


Fig. 7. Dominator Tree

an interesting feature of the dominator tree of a call graph is that independent loops dominated by a function are represented as different branches out of the dominating function which precludes the need to dissolve simple loops. Dominator trees, therefore, can be used as the basis of a graph used to propagate free variables. Since dominator trees can be computed in linear time [16], the need to repeatedly compute strongly connected subcomponents, which makes the MM-algorithm cubic, can be eliminated to reduce the complexity of lambda lifting.

The dominator tree, however, does not capture all dependencies between functions needed for lambda lifting. We classify these missing dependencies as *vertical* and *horizontal* dependencies, described in Sections 5 and 6 respectively. Vertical dependencies capture dependencies arising due to recursion between ancestors and descendants in the dominator tree. Horizontal dependencies capture dependencies arising between functions that do not have a vertical dependence in the dominator tree. Vertical dependencies are annotated on the dominator tree and are used to drive the propagation of free variables throughout the tree. Horizontal dependencies are added to the dominator tree, which necessitates coalescing the strongly connected components to obtain a directed acyclic graph. The resulting coalesced graph is used to propagate free variables.

5 Vertical Function Dependencies

We define a *downward* vertical dependence as the dependence that exists between a function and a descendant in the parse tree. At runtime, a call to any local function, g, must be preceded by calls to g's ancestors in the parse tree which are also ancestors of g in the dominator tree of the call graph of the program. Any extraneous parameters that g contributes to its ancestors can be propagated up the dominator tree.

We define an *upward* vertical dependence as the dependence that exists between a function, g, and a function, f, which is an ancestor of g. The function g may depend on several of its ancestors in the dominator tree of which we are interested in the one that has the maximum depth. We define the lowest upward vertical dependence of g, LD_g, as the function with the maximum depth in the

```
(define (f x y z)
  (define (g ...) (...x...i...))
  (define (h ...) (...y...f...))
  (define (i ... (...z...f...g...))
 (...g...h...))
```

Fig. 8. Fourth Sample Scheme Pseudo-code

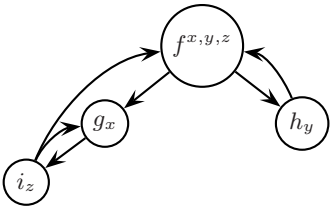


Fig. 9. Call Graph and Relevant Variables

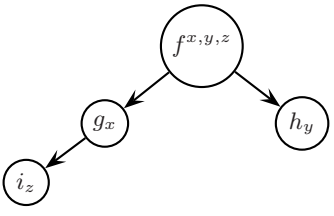


Fig. 10. Dominator Tree (DT)

dominator tree that g depends on. LD_g, if it exists, is the ancestor of g with the maximum depth that is either called by g or is called from any of g's descendants in the dominator tree. For example, consider the pseudo-code in Figure 6. The dominator tree for this code is displayed in Figure 7. Observe that i calls g and h which are ancestors of i in the dominator tree. Since h has the maximum depth, we have that $LD_i = h$. The function h does not call any of its ancestors, but it depends on its ancestor g which is called from i. Since g is the only ancestor of h that is referenced by any function in the subtree rooted at h, we have that $LD_h = g$. Finally, $LD_g = LD_f = \emptyset$ because none of the ancestors of g or f are referenced by functions in the subtrees of the dominator tree rooted at these functions.

To start exploring lambda lifting algorithms let us restrict our observations to the class of programs in which all dependencies are vertical (this restriction will be removed in the next section). Upward vertical dependence is not captured by a dominator tree, but can be computed as free variables are propagated up the dominator tree to identify the extraneous parameters contributed by downward vertical dependencies. Along with free variables, the set of referenced ancestor functions is propagated up the dominator tree.

Clearly, all extraneous parameters for g contributed by its descendants will reach g during an upward propagation. The following theorem establishes that LD_g, if it exists, contains all the extraneous parameters needed by g that are contributed by its ancestors in the dominator tree.

Theorem 1. *The set of extraneous parameters needed by LD_g contains all the extraneous parameters needed by g from its ancestors.*

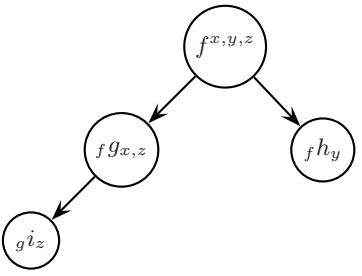


Fig. 11. DT After Upward Propagation

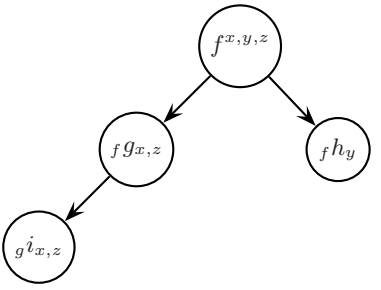


Fig. 12. DT After Downward Propagation

Proof. Let DT be the dominator tree for a call graph, CG, and let h be LD_g*. Assume x is an ancestor-contributed extraneous parameter needed by g that is not a member of the set of extraneous parameters needed by h. If x is defined by an ancestor of h, then x must be a member of the set of extraneous parameters needed by h which contradicts our assumption. This follows from observing that h must carry x in order to make it available to g. If x is defined by h or a descendant of h, then there must exist a path in CG from g to a function where x is a known free variable that does not contain the function that declares x. All the functions on this path must be descendants of h in the dominator tree which means that* $LD_g \neq h$*. This contradicts our assumption and completes the proof that x must be a member of the set of extraneous parameters needed by h. Q.E.D*

To illustrate how vertical dependencies are used in lambda lifting consider the pseudo-code in Figure 8 whose call graph is displayed in Figure 9. Figure 10 displays its dominator tree. Free variables needed by functions due to downward vertical dependence can be propagated up the dominator tree using a depth-first traversal. After this is done, the variable z has been propagated from i to g. In addition during this propagation step, the LD_i of each function i is computed by also propagating relevant upward vertical dependencies. LD_i is g and LD_h is f, because for leaves the LD function is the lowest ancestor in the dominator tree that they directly reference. Nodes pass the set of referenced ancestors back up the tree along with their free variables. In this manner, LD_g becomes f as it is the ancestor of g with the largest depth that is referenced from a function in the subtree rooted at g. The result of this step is displayed in Figure 11 in which the subscript to the left of each function name is its lowest dependence function. Finally, free variables need to be propagated down the dominator tree to satisfy upward vertical dependencies. This propagation proceeds in a breadth-first order propagating to function i the free variables needed by LD_i. A breadth-first order propagation is required to guarantee that the extraneous parameters of ancestor functions are known before the extraneous parameters of any descendant function are computed (which satisfies the key lesson highlighted from the J-algorithm). During this step the variable x is propagated from g to i. The result of this propagation step is displayed in Figure 12.

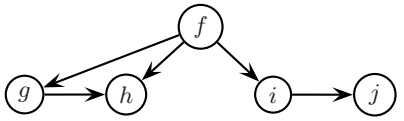


Fig. 13. Call Graph with Calls Among Siblings

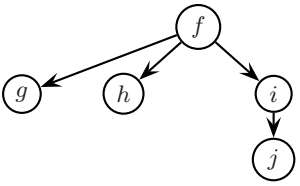


Fig. 14. Dominator Tree Lacks Some Function Dependencies

6 Horizontal Function Dependencies

Lexical scoping restricts the set of functions that may be directly referenced by any given function to itself, its children, its ancestors, its siblings, and its uncles in the parse tree. References to itself do not contribute new free variables to the lifted version of the function. References to ancestors and children are all captured as vertical dependencies annotated in the dominator tree as described in the previous section. References to siblings and uncles are references to functions with which there may be no dominance relation. For example, consider the call graph in Figure 13. Assume that f is the parent of g, h, i, and j in the parse tree. The dominator tree is displayed in Figure 14. Notice that i dominates its sibling j while there is no dominance relation between g and h despite g having a reference to h. The dependence of g on h, in fact, is not captured by the dominator tree.

We define a *horizontal* dependence as a reference to a function that is not an ancestor or a descendant in the dominator tree (i.e. a reference to a sibling or an uncle in the parse tree). The free variables of a horizontal dependence must also be propagated from the callee to the caller. Since horizontal dependencies are not captured by the dominator tree of a call graph, a dominator tree must be augmented into a graph to capture horizontal dependencies.

To convert a dominator tree into a graph that captures horizontal dependencies, the dominator tree is augmented with the edges between functions in the call graph that do **not** have a vertical dependence. We call this graph an EDT (**E**xtended **D**ominator **T**ree) graph and the new edges are called lateral edges. If the resulting EDT graph does not contain any cycles then it only has *simple* horizontal dependencies. Otherwise, it has *complex* horizontal dependencies. Clearly, the EDT graph for a program that only has functions with vertical dependencies is its annotated dominator tree.

First, we highlight some important properties of EDT graphs. Second, we extend our lambda lifting algorithm to handle the class of programs that have simple horizontal dependencies. Finally, we extend our lambda lifting algorithm to handle arbitrary programs that may contain complex horizontal dependencies.

6.1 Important Properties of EDT Graphs

Formally, the set of lateral edges, E_l, in an EDT graph formed from the dominator tree, DT, of a call graph, CG, is defined as:

```
(define (f x)
  (define (g ...) (...x...a...b...c...d...))
  (define (a ...) (...b...))
  (define (b ... (...c...d...))
  (define (c ...) (...g...))
  (define (d ...) (...))
 (...g...))
```

Fig. 15. Fifth Scheme Pseudo-code

$$E_l = \{(f, g) \in CG \mid f \text{ and } g \text{ do not have a dominance relation}\}.$$

The set E_l endows the EDT graph with important properties outlined by the following theorems. After establishing the validity of these properties we will point out their significance for lambda lifting.

Theorem 2. *If (f,g) $\in E_l$, then the parent of g, p_g, in the dominator tree, DT, dominates f.*

Proof. Let G be the EDT graph obtained by only extending DT with the lateral edge from f to g and let r be the root function of DT. If there is a path in G from r to g that contains f and that does not contain p_g, then p_g does not dominate all paths from r to g. This means that DT can not be the dominator tree which contradicts our assumption. Q.E.D.

Having established that the parent of the called function for a lateral edge in the EDT graph dominates the caller, we can now establish that all the ancestors of the called function dominate the caller. The proof simply exploits the fact that domination is a transitive property.

Theorem 3. *If (f,g) $\in E_l$, then all ancestors of g in the dominator tree, DT, dominate f.*

Proof. Theorem 2 establishes that the parent of g dominates f. All other ancestors of g dominate its parent. Therefore, all of g's ancestors dominate f. Q.E.D.

The significance of Theorems 2 and 3 for lambda lifting is that the existence of a lateral edge from f to g in an EDT graph means that LD_g, if it exists, dominates f. Therefore, LD_g may also be LD_f. This occurs when none of the nodes in the dominator tree path from the parent of g to f are LD_f. In addition to free variables, LD information must be propagated from callees to callers across lateral edges.

6.2 Simple Horizontal Dependencies

When an EDT graph only has simple horizontal dependencies (i.e. there are no strongly connected compinents in the EDT graph) it suffices to first propagate free variables and lowest dependence information between functions using

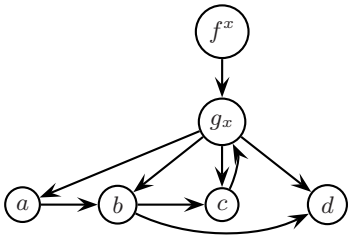


Fig. 16. Call Graph

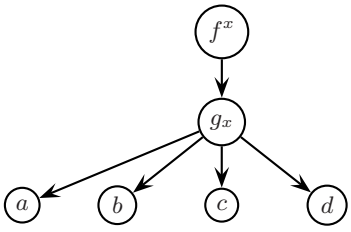


Fig. 17. Dominator Tree

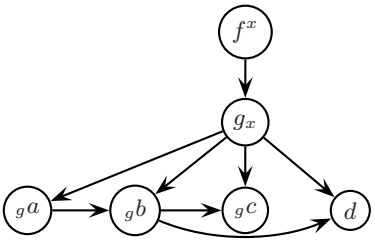


Fig. 18. After Depth-First Propagation

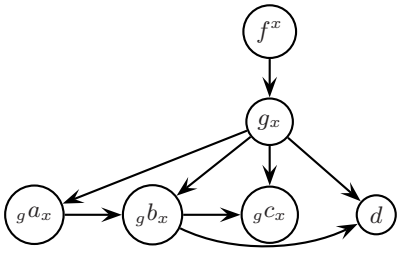


Fig. 19. After Breadth-First Propagation

a depth-first traversal (akin to propagating up the dominator tree) and then to propagate free variables in breadth-first order exploiting lowest dependence information (akin to propagating down the dominator tree). The correctness of the second propagation follows from observing that free variables are propagated from callees to callers and from Theorem 3 that guarantees lowest dependence information can safely be propagated across lateral edges.

To illustrate the use of horizontal dependence information in the absence of strongly connected components consider the pseudo-code in Figure 15 and its call graph in Figure 16. The dominator tree for this graph is displayed in Figure 17. Extending the dominator tree with edges between functions that do not have a vertical dependence results in the original call graph without the edge from c to g. Figure 18 displays the results of propagating free variables and LD information after a depth-first traversal. The node representing c has no successors and, therefore, LD_c is g (the lowest ancestor it references). No free variables propagate between the functions in this step, but LD_c, g, propagates to become LD_b and LD_a. Figure 19 displays the results of propagating free variables in a breadth-first order by exploiting LD information. Each function receives the free variables of its LD function. That is, a, b, and c receive x.

6.3 Complex Horizontal Dependencies

The augmentation of the dominator tree, however, may lead to an EDT graph that is no longer acyclic. That is, the resulting graph may contain strongly connected components. This occurs, for example, when two siblings in the dominator

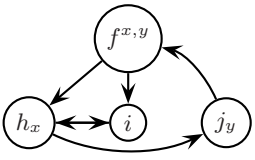


Fig. 20. Call Graph

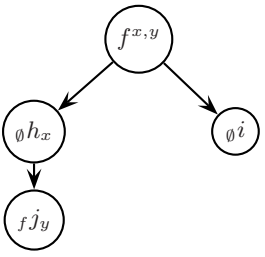


Fig. 21. Dominator Tree

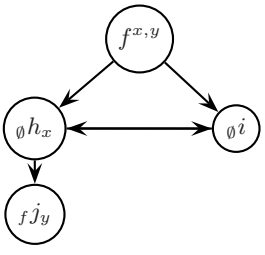


Fig. 22. EDT Graph

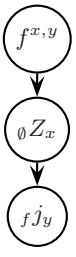


Fig. 23. Coalesced EDT Graph

tree are mutually recursive. In the presence of strongly connected components, it no longer suffices to simply propagate free variables and LD information using a depth-first traversal. The problem is that such a traversal does not guarantee that all successors of a node are processed first.

Strongly connected components must be coalesced, but as learned from the MM-algorithm sets of functions that include a dominating function and the functions it dominates should not be coalesced. That is, functions that have a vertical dependence should not be coalesced. This observation suggests that within a strongly connected component only functions at the same level in the dominator tree can be coalesced together. Notice that a function at level n in the dominator tree can not declare any variables that are free in other functions at level n. This means that they do not have a dominance relation and it is safe to coalesce these functions together, because none of these functions will unnecessarily contribute free variables to each other.

The goal, therefore, is to coalesce strongly connected components in an EDT graph without loosing vertical dependence information. To achieve this it is helpful to distinguish between two types of edges in an EDT graph. The first kind of edge is a *simple lateral edge* which occurs between functions at the same level of the dominator tree (i.e. edges between siblings in the dominator tree). Any strongly connected components formed solely by simple lateral edges can be coalesced in the EDT graph, because among the siblings in each component there is no dominating function. If an EDT graph is created by solely adding simple lateral edges to the dominator tree, then after coalescing strongly connected

components the EDT graph is acyclic. Thus, lambda lifting can proceed as described in section 6.2 by making a coalesced node's free variables the union of the free variables of the functions in the strongly connected component and by making the node's LD function be the $max_f(LD_g)$, where g is a function in the strongly connected component. To illustrate this concept consider the call graph in Figure 20. Its dominator tree, displayed in Figure 21, reflects the known facts after its creation: i and h have no known upward vertical dependencies, LD_j is f, x is free in h, and y is free in j. The EDT graph, displayed in Figure 22, is created by adding the two lateral edges between i and j in the dominator tree. The strongly connected component formed by $\{h, i\}$ is coalesced into a node, say, Z. The set of free variables of Z is $\{x\}$ and LD_Z is $\emptyset$. The result of this transformation is displayed in Figure 23. After the depth-first propagation the set of free variables of Z is $\{x, y\}$ and $LD_Z = f$. Nothing propagates during the breadth-first propagation (because f has no free variables). After the propagation steps, we have that $\{x, y\}$ are the required free variables for h and i which is precisely what is needed.

The second kind of edge is an *upward lateral edge* which exists between functions at different levels of the dominator tree. These edges always occur from a node for a function, g, at level n to a node for function, f, at level $n - i$, where $i \geq 1$, such that f is not an ancestor of g in the dominator tree[3]. The existence of such an edge, means that g needs the free variables of f. Notice, however, that f may not need all of g's free variables despite being in the same strongly connected component. The free variables of g not needed by f are those that are local to the strongly connected component and that are not lexically visible nor declared by f. All of these variables must be declared by a function with a depth greater than or equal to the depth of f in the dominator tree.

Notice that the set of functions in the strongly connected component may include siblings of f in the dominator tree. The incoming upward lateral edge to f means that these siblings need the same free variables. This follows from observing that they all need as free variables the variables declared by common ancestors in the dominator tree that are free in the strongly connected component. Therefore, we have that the siblings of a function in the dominator tree, like f that has an incoming upward lateral edge, that are in the same strongly connected component can be coalesced with f without local free variables being unnecessarily propagated during lambda lifting. Coalescing only siblings in a strongly connected component preserves vertical dependence information and provides a directed acyclic graph that can be used to compute the free variables needed by each function in an arbitrary program.

To illustrate the use of horizontal dependence information in the presence of strongly connected components created by upward lateral edges consider the pseudo-code in Figure 24 and its call graph in Figure 25. Its dominator tree is displayed in Figure 26. The first step is to extend the dominator tree with simple lateral edges. The resulting graph is displayed in Figure 27. Five simple

[3] There can not exist any edges in the other (i.e. downward) direction from f to g in a dominator tree.

```
(define (f x)
  (define (g ...) (...h...))
  (define (h c) (...j...k...)))
    (define (j ...) (k...c...l...m...)))
    (define (k a b) (...j...n...o...)))
      (define (n ...) (...a...o...)))
      (define (o ...) (...b...i...)))
  (define (i ...) (...h...))
  (define (l ...) (...g...))
  (define (m ...) (...x...))
 (...g...h...i...)))
```

Fig. 24. Sixth Scheme Pseudo-code.

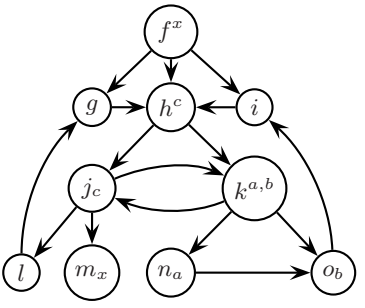


Fig. 25. Call Graph

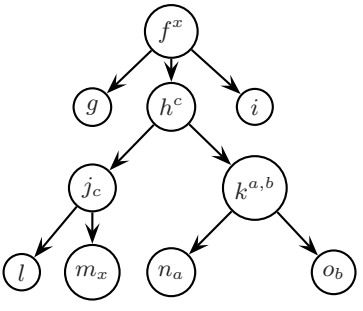


Fig. 26. Dominator Tree

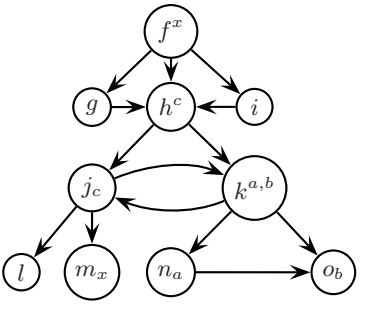


Fig. 27. DT with Simple Lateral Edges

lateral edges have been added to extend the dominator tree. These additions have formed a strongly connected component that contains the functions j and k. These functions are coalesced to form a new node S. The set of free variables for S is obtained from the union of the free variables of j and k. The resulting graph is displayed in Figure 28. The graph in Figure 28 is now extended with upward lateral edges. If a function on either side of an edge has been coalesced then the coalesced node replaces the function. The result of this extension adds edges from l to g and from o to i. The result is displayed in Figure 29. This graph now has a strongly connected component formed by $\{g, h, i, S, l, n, o\}$. Function g has an incoming upward lateral edge and, therefore, it is coalesced with its siblings h and i that are also members of the strongly connected component. Given that i, a function with an incoming lateral edge, has been coalesced there is no need for further action with it. No other functions have an incoming upward lateral edge which means the graph is now acyclic. The finalized EDT graph is displayed in Figure 30 in which Q represents the coalesced functions $\{g, h, i\}$. This graph can now be used to propagate free variables and LD information as done in section 6.2.

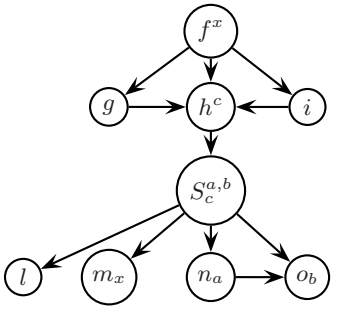


Fig. 28. Graph After First Coalescing Step

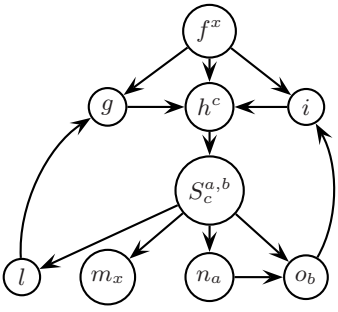


Fig. 29. Added Upward Lateral Edges

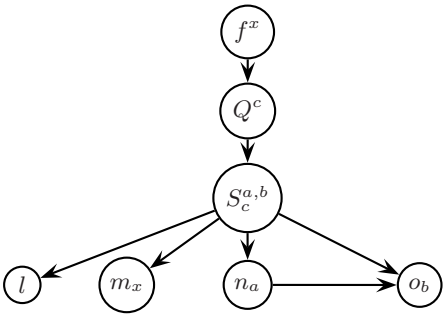


Fig. 30. Completed EDT Graph After Second Coalescing Step

7 The Algorithm, Complexity, and Correctness

7.1 The New Lambda Lifting Algorithm

The new lambda lifting algorithm builds a directed acyclic EDT graph from the call graph of a program. Propagation of free variables then proceeds in two steps: the first using a depth-first traversal and the second using a breadth-first traversal. The steps in the algorithm can be outlined as follows:

1. Build the call graph, CG, of the program from its parse tree.
2. Build the dominator tree, DT, for CG.
3. Extend DT with simple lateral edges and coalesce strongly connected components to obtain an acyclic graph EDT'.
4. Extend EDT' with upward lateral edges and compute strongly connected components. Coalesce functions that have an incoming upward lateral edge with their dominator tree siblings that are members of the same component. The resulting graph is the directed acyclic EDT'' graph.
5. Use EDT'' to propagate free variables and LD information using a depth-first traversal.
6. Use EDT'' to propagate free variables using LD information using a breadth-first traversal.

7. For each function, f, make f scope insensitive by adding its complete set of free variables as parameters to f and as arguments to each reference to f.
8. Remove block structure by floating each function to the global level.

7.2 Complexity and Correctness

For a program P, let i be the number of functions, let e be the number of function calls, let v be the number of variables declared, and let n be the size of the program (i.e. $i + e + v$). Step 1 is proportional to $O(e + i)$ or simply $O(n)$. Step 2 is $O(n)$ [16]. For steps 3 and 4 extending a graph with edges is $O(e+i)$ or simply $O(n)$. The computation of strongly connected components is $O(n)$ [5,6] and their coalescing is $O(n^2)$. For step 5, the propagation of free variables and LD information is $O(e * (i + v))$, or simply $O(n^2)$, assuming the union operation is done in linear time. A similar line of reasoning holds for step 6. Step 7 is $O(v + i + e)$ or simply $O(n)$. Finally, step 8 is $O(n)$. This means that optimal lambda lifting is done in $O(n^2)$. Since lambda lifting can generate an output program of size $O(n^2)$, the time complexity of this algorithm is optimal [3].

We have not formally proven the correctness of the presented algorithm and we only argue informally for its correctness. The correctness of the algorithm hinges on correctly computing the set of required variables for each function. The required set of free variables for a function, f, depends on the free variables f directly references and on a subset of the free variables transitively needed by the functions f calls. The computation of the latter subset is achieved by never coalescing a dominating function with any functions it dominates. This leads to a graph in which the breadth-first propagation in Step 6 completes the computation of the required free variables of any ancestor function in the parse tree before any successor function as done in the J-algorithm and prevents free variables local to a strongly connected component to be unnecessarily propagated. The required set of free variables computed for each function is complete, because all functional dependencies are captured by the EDT graph. Lateral and downward vertical dependencies are captured by edges and upward vertical dependencies are captured by LD information.

8 Concluding Remarks

This article presents an optimal graph-based $O(n^2)$ lambda lifting algorithm. The algorithm is optimal in the sense that it computes the minimal set of free variables required by each function to make them scope insensitive. The algorithm is also asymptotically optimal, because a lambda lifted program of size $O(n^2)$ is computed in $O(n^2)$ steps. The new algorithm is superior to Johnsson's and to Morazán's and Mucha's algorithms by reducing the complexity of optimal lambda lifting from $O(n^3)$ to $O(n^2)$ and it is superior to Danvy's and Schultz's algorithm by being optimal. Nonetheless, this new algorithm owes a great deal of its creation to these predecessors. Considering that Johnsson's original algorithm first appeared in 1985, this newest algorithm has been over 20 years in the

making. It is, indeed, a tribute to all these algorithms and to the work of the cited authors from whom we borrowed ideas and inspiration.

Free variables arise in programs due to the nesting of function definitions. This suggests that identifying free variables may rely heavily on lexical analysis. The algorithm presented, however, only relies on lexical analysis to identify the free variables directly referenced by each function and the function dependencies of each function. Once this lexical information is known, the algorithm builds and relies on non-lexical elements such as the call graph, the dominator tree, and sets of free variables. In essence, lambda lifting is not solely an exercise in lexical analysis.

As part of our future work we are interested in testing the runtime efficiency of the different algorithms to determine their impact on compilation time. Although compilers only spend a small amount of time on lambda lifting, such testing will determine the practical impact of this work. Future work also includes the implementation of a closureless functional language that uses applicative-order evaluation. The main idea behind the design of this new langauge is to dynamically generate functions that are specialized based on the bindings of its free variables instead of allocating closures [13]. Lambda lifting identifies for us the variables that are used to specialize functions.

Acknowledgements

The authors thank Olivier Danvy, Sven-Bodo Scholz, and Barbara Mucha for the discussions during and after IFL 2005 that initiated us down the path that lead to the new algorithm presented in this article. Marco T. Morazán also thanks TLTC at Seton Hall University for the support received through a Faculty Innovation Grant.

References

1. Consel, C.: A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages. In: Proc. of the Symp. on Partial Evaluation and Semantics-Based Program Manipulation, June 1993, pp. 145–154. ACM Press, New York (1993)
2. Danvy, O., Schultz, U.P.: Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. Theoretical Computer Science 248(1–2), 243–287 (2000)
3. Danvy, O., Schultz, U.P.: Lambda-Lifting in Quadratic Time. Journal of Functional and Logic Programming 2004(1) (July 2004)
4. Friedman, D.P., Wand, M., Haynes, C.T.: Essentials of Programming Languages. The MIT Press, Cambridge (2001)
5. Gibbons, A.: Algorithmic Graph Theory. Cambridge University Press, Cambridge (1985)
6. Gould, R.: Graph Theory. The Benjamin/Cummings Publishing Company, Inc. (1988)
7. Matthews, J., Findler, R., Graunke, P., Krishnamurthi, S., Felleisen, M.: Automatically Restructuring Programs for the Web. Automated Software Engineering 11(4), 337–364 (2004)

8. Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations. In: Proc. of a Conf. on Functional Prog. Lang. and Comp. Arch., pp. 190–203. Springer, New York (1985)
9. Johnsson, T.: Target Code Generation from G-Machine Code. In: Fasel, J.H., Keller, R.M. (eds.) Graph Reduction: Proceedings of a Workshop at Santa Fé, New Mexico, New York, NY, pp. 119–159. Springer, Heidelberg (1987)
10. Jones, S.L.P.: The Implementation of Functional Programming Languages. Prentice-Hall International Series in Computer Science. Prentice-Hall, Upper Saddle River (1987)
11. Jones, S.L.P., Lester, D.: A Modular Fully-lazy Lambda Lifter in HASKELL. Software - Practice and Experience 21(5), 479–506 (1991)
12. Jones, S.L.P., Lester, D.: Implementing Functional Languages: A Tutorial. Prentice Hall International Series in Computer Science (1992)
13. Morazán, M.T.: Towards Closureless Functional Languages. In: Arabnia, H. (ed.) Proc. of the Int. Conf. on Prog. Lang. and Compilers, pp. 57–63. CSREA Press (2005)
14. Morazán, M.T., Mucha, B.: Improved Graph-Based Lambda Lifting. In: Arabnia, H. (ed.) Proc. of the Int. Conf. on Prog. Lang. and Compilers, June 2006, pp. 896–902. CSREA Press (2006)
15. Oliva, D.P., Ramsdell, J.D., Wand, M.: The VLISP Verified PreScheme Compiler. Lisp and Symbolic Computation 8(1-2), 111–182 (1995)
16. Alstrup, S., Harel, D., Lauridsen, P.W., Thorup, M.: Dominators in Linear Time. SIAM Journal on Computing 28(6), 2117–2132 (1999)

Appendix

Tree. A *tree* is a connected, directed, and acyclic graph in which there is a node, called the *root*, such that there is a path from the root to every other node in the graph. The root node has no incoming edges and all other nodes have only one incoming edge. When an edge from node A to node B exists, we say that A is the parent of B. The *depth* or *level* of a node N in a tree rooted at R is the number of edges in the path from R to N. The *ancestors* of N are all the nodes on the path from R to the parent of N. The *descendants* of N are all the nodes that are reachable from the subtree rooted at N.

Dominator Tree. In a graph G with root node R, a node N *dominates* a node M if every path from R to M must pass through N. The *immediate dominator* of a node M is a node N if $\nexists K \in G$: N dominates K and K dominates M. The *dominator tree* T of G contains the same set of nodes as G and has an edge from a node N to a node M when N is the immediate dominator of M.

Strongly Connected Component. The nodes $N_1 \ldots N_k$ of a directed graph G form a *strongly connected component* if for every pair of distinct nodes, N_i and N_j, there is a path from N_i to N_j and a path from N_j and N_i.

The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra

Department of Information and Computing Sciences,
Universiteit Utrecht,
P.O. Box 80.089,
Padualaan 14, Utrecht, Netherlands
{atze,jeroen,doaitse}@cs.uu.nl
http://www.cs.uu.nl

Abstract. In this paper we describe the structure of the Essential Haskell Compiler (EHC) and how we manage its complexity, despite its growth from essentials to a full Haskell compiler. Our approach splits both language and implementation into smaller, manageable steps, and uses specific tools to generate parts of the compiler from higher level descriptions.

1 Introduction

Haskell is a perfect example of a programming language which offers many features improving programming efficiency by offering a sophisticated type system. As such it is an answer for the programmer looking for a programming language which does as much as possible of the programmer's job, while at the same time guaranteeing program properties like "well-typed programs don't crash". However, the consequence is that a programming language implementation is burdened by these responsibilities, and consequently becomes quite complex. Haskell thus also is a perfect example of a programming language for which compilers are complex. Testimony to this observation is the Glasgow Haskell Compiler (GHC) [31,32,35,37], which simultaneously incorporates many novel features, is used as a reliable workhorse for many a functional programmer, and offers a research platform for language designers. As a result, modifying GHC requires much knowledge of GHC's internals.

In this paper we show how we deal with the complexity of compiling Haskell in the Essential Haskell (EH) Compiler (EHC) [14,15]. EH intends

- to compile full Haskell (the H in EH)
- to offer an implementation in terms of the essential, or desugared, core language constructs of Haskell (the E in EH)
- to provide a solid framework for research (i.e., extendable for experimentation) and education (another interpretation of the E in EH)

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 57–74, 2008.

In particular the following areas require attention:

- **Implementation complexity.** (Section 2) The amount of work a compiler has to do is a source of complexity. We organise the work as a series of smaller transformation steps [35,39] between various internal representations.
- **Description complexity.** (Section 3) The specification of parts of the implementation itself can become complex because low-level details are visible. We use domain specific languages which factor out such low-level details, so they are dealt with automatically.
- **Design complexity.** (Section 4) Experiments with language features are usually done in isolation. We describe their implementation in isolation, as a sequence of language variants, building on top of each other.
- **Maintenance complexity.** (Section 5) Actual compiler source, its documentation, and its specification tend to become inconsistent over time. We fight such inconsistencies by avoiding their main cause: duplication. Whenever two artefacts have to be consistent, we generate them from a common description.

In the next sections we explain how we deal with each of these complexities.

2 Coping with Implementation Complexity: Transform

EHC is organised as a sequence of transformations between internal representations of the program being compiled. In order to keep the compiler understandable, we keep the transformations simple, and consequently, there are many. This approach is similar to the one taken in GHC [35,36,38]. All our transformations are expressed as a full tree walk over the data structure, using a tool for easily defining tree walks (see Section 3.1). At each step in which the representation changes drastically we introduce a separate data structure (or "language"). Fig. 1 shows these languages and the transformations between them:

- **HS** (Haskell) is a representation of the program text as parsed. It is used for desugaring, name and dependency analysis, and making binding groups explicit.
- **EH** (Essential Haskell) is a simplified and desugared representation. It is used for type analysis and code expansion of class system related constructs.
- **Core** is a representation in an untyped λ-calculus.
- **Grin** (Graph reduction intermediate notation) is a representation proposed by Boquist [12,13] in which local definitions have been made sequential and the need for evaluation has been made explicit.
- **Silly** (Simple imperative little language) is a simple abstraction of an imperative language with an explicit stack and heap, and functions which can be called and tail-called.
- **C** is used here as a universal back-end, hiding the details of the underlying machine. Primitive functions are implemented here.

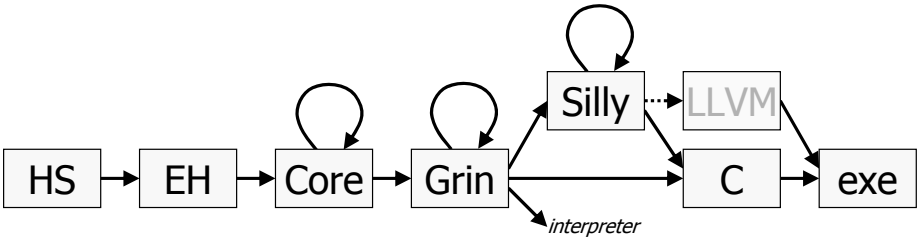


Fig. 1. Intermediate languages and transformations in the EHC pipeline

– **LLVM** (Low level virtual machine) is an imperative language which, other than C, is *intended* to be a universal back-end [29]. We have it under consideration as an alternative route to attain executable code.

As can be seen from the figure, the compilation pipeline branches after the Grin stage, offering different modes of compilation:

– Grin code can be interpreted directly by a simple (and thus slow) interpreter.
– Grin code can be translated to C directly. In this mode, the program is represented in a custom bytecode format, stored in arrays, and executed by an interpreter written in C. Its speed is comparable to that of Hugs [1].
– Grin code can be translated to executable code via transformations which perform global program analysis, and generate optimized Silly code, which can be further processed through either the C or LLVM route.

The transformations between the languages mentioned above bring the program stepwise to a lower level of representation, until it can be executed directly. Most of the simplification work however is done by the transformations that are indicated by a loop in Fig. 1, i.e., for which the source and target language are the same. We strive to have many small transformations rather than a few complicated ones. To give an idea, we list a short description of the more important of these transformations. Some of these are necessary simplifications, others are optimisations that can be left out.

– Transformations on the Core language include:
  • Cleanup transformations: *Eta-reduction*, *Eliminating trivial applications*, *Inline let alias*, *Remove unnecessary letrec mutual recursion*
  • *Constant propagation* and *Rename identifiers to unique names*
  • Lambda lifting, split up in: *Full laziness of subexpressions*, *Lambda/CAF globals passed as argument*, *Float lambda expressions to global level*
– Transformations on the Grin language include:
  • Transformations on separate modules: *Alias elimination*, *Unused name elimination*, *Eval elimination*, *Unboxing*, *Local inlining*
  • Transformations based on a global abstract interpretation that determines possible constructors of actual parameters: *Inline eval operation*, *Remove dead case alternatives and unused functions*, *Global inlining*

  - Transformations that remove higher-level constructs, such as splitting complete nodes into their constituent fields.
- Transformations on the Silly language include:
  - *Shortcut*: avoid unnecessary copying of local variables
  - *Embed*: map local variables to stack positions

3 Coping with Description Complexity: Use Tools

Haskell is well suited as an implementation language for compilers, among others because of the ease of manipulating tree structures. Still, if one needs to write many tree walks, especially if these involve multiple passes over complicated syntax trees, the necessary mutually recursive functions tend to become hard to understand, and contain large pieces of boilerplate code. In the implementation of EHC we therefore use a chain of preprocessing tools, depicted in Fig. 2.

We use the following preprocessing tools:

- **UUAGC** (Utrecht University Attribute Grammar Compiler), which enables us to specify abstract syntax trees and tree walks over them using an attribute grammar (AG) formalism [9,15,16,44].
- **Shuffle**, which deals with the compiler organisation and logistics of many different language features, and provides a form of literate programming.
- **Ruler**, a translator for an even more specialized language than AG, which enables a high-level specification of type inferencing, generating both AG code and LaTeX documentation [17].

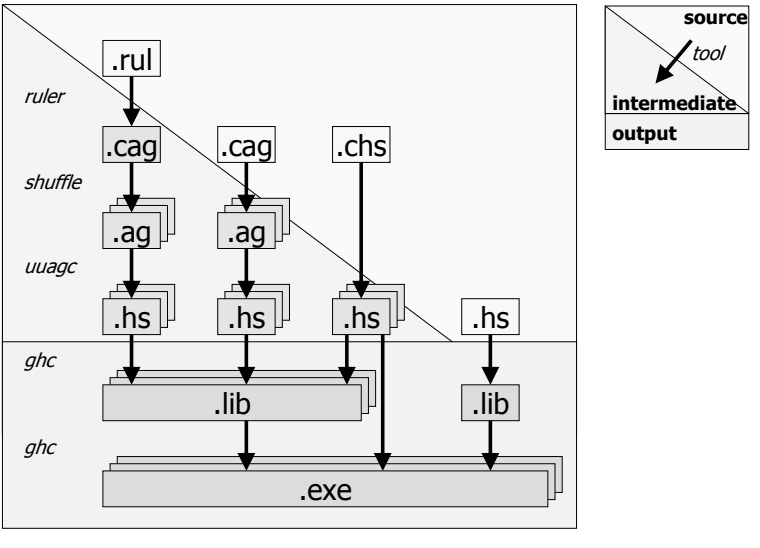


Fig. 2. Chain of tools used to build EHC

In the remainder of this section we elaborate on the rationale of UUAGC (Section 3.1) and *Ruler* (Section 3.2). We illustrate their use with example code, which implements part of a Hindley-Milner type checker. In the section on UUAGC this is idealized toy code, but in the section on *Ruler* we show actual code taken from EHC for the same example. In Section 4 and 5 we continue with the rationale and use of *Shuffle*.

3.1 UUAGC, a System for Specifying Tree Walks

Higher-order functional languages are famous for their ability to parameterize functions not only with numbers and data structures, but also with functions and operators. The standard textbook example involves the functions *sum* and *product*, which can be defined separately by tedious inductive definitions:

$$
\begin{array}{lll}
sum & [\,] & = 0 \\
sum & (x : xs) & = x + sum\ xs \\
product & [\,] & = 1 \\
product & (x : xs) & = x * product\ xs
\end{array}
$$

This pattern can be generalized in a function *foldr* that takes as additional parameters the operator to apply in the inductive case and the base value:

$$
\begin{array}{lll}
foldr\ op\ e & [\,] & = e \\
foldr\ op\ e & (x : xs) & = x\ `op`\ foldr\ op\ e\ xs
\end{array}
$$

Once we have this generalized function, we can partially parameterize it to obtain simpler definitions for *sum* and *product*, and many other functions as well:

$$
\begin{array}{ll}
sum & = foldr\ (+)\ 0 \\
product & = foldr\ (*)\ 1 \\
concat & = foldr\ (+\!\!+)\ [\,] \\
sort & = foldr\ insert\ [\,] \\
transpose & = foldr\ (zipWith\ (:))\ (repeat\ [\,])
\end{array}
$$

The idea that underlies the definition of *foldr* (capturing the pattern of an inductive definition by adding a function parameter for each constructor of the data structure), can also be used for other data types, and even for multiple mutually recursive data types. Functions that can be expressed in this way are called *catamorphisms* by Bird, and the collective extra parameters to *foldr*-like functions *algebras* [10,11]. Thus, $((+), 0)$ is an algebra for lists, and $((+\!\!+), [\,])$ is another. In fact, every algebra defines a *semantics* of the data structure.

Outside circles of functional programmers and category theorists, an algebra is simply known as a "tree walk". In compiler construction, algebras could be very useful to define a semantics of a language, or bluntly said to define tree walks over the parse tree. This is not widely done, due to the following problems:

1. Unlike lists, which have a standard function *foldr*, in a compiler we deal with (many) custom data structures to describe the abstract syntax of a

language, so we have to invest in writing a custom *fold* function first. Moreover, whenever we change the abstract syntax, we need to change the *fold* function, and every algebra.
2. Generated code can be described as a semantics of the language, but often we need an alternative semantics: pretty-printed listings, warning messages, and various derived structures for internal use (symbol tables etc.). This can be done in one pass by having the semantic functions in the algebra return tuples, but this makes them hard to handle.
3. Data structures for abstract syntax tend to have many alternatives, so algebras end up to be clumsy tuples containing dozens of functions.
4. In practice, information not only flows bottom-up in the parse tree, but also top-down. E.g., symbol tables with global definitions need to be distributed to the leaves of the parse tree to be able to evaluate them. This can be done by making the semantic functions in the algebra higher order functions, but this pushes the handling of algebras beyond human control.
5. Much of the work is just passing values up and down the tree. The essence of a semantics in the algebra is obscured by lots of boilerplate.

In short: the concepts of catamorphism and algebra apply here, but their encoding in Haskell is cumbersome and becomes prohibitively complex. Many compiler writers thus end up writing ad hoc recursive functions instead of defining the semantics by an algebra, or even resort to non-functional techniques. Others try to capture the pattern using monads [33]. Some succeed in giving a concise definition of a semantics, often using proof rules of some kind, but loose executability. For the implementation they still need conventional techniques, and the issue arises whether the program soundly implements the specified semantics.

To save the nice idea of using an algebra for defining a semantics, we use a preprocessor for Haskell [44] that overcomes the mentioned problems. It is not a separate language; we can still write auxiliary Haskell functions, and use all abstraction techniques and libraries. The preprocessor just allows a few additional constructs, which are translated into custom *fold*-like functions and algebras.

We describe the main features of the preprocessor here, and explain why they overcome the five problems mentioned above. For a start, the grammar of the abstract syntax of the language is defined in **data** declarations, which are like a Haskell **data** declaration with named fields, except that we do not have to write braces and commas and that constructor function names need not be unique. As an example, we show a fragment of EHC that represents a lambda calculus:

```
data Expr
   = Var  name :: Name
   | Let  decl :: Decl  body :: Expr
   | App  func :: Expr  arg  :: Expr
   | Lam  arg  :: Pat   body :: Expr
data Decl
   = Val  pat :: Pat  expr :: Expr
data Pat
```

```
= Var  name :: Name
| App  func :: Expr  arg :: Expr
```

The preprocessor generates corresponding Haskell **data** declarations (making the constructors unique by prepending the type name, like *Expr_Var*), and more importantly, generates a custom *fold* function. This overcomes problem 1.

For any desired value we wish to compute from a tree, we can declare a "synthesized attribute" (the terminology goes back to Knuth [26]). Attributes can be defined for one or more data types. For example, we can define that for all three datatypes we wish to synthesize a pretty-printed listing, and that expressions in addition synthesize a type and a variable substitution map:

```
attr Expr Decl Pat syn listing :: String
attr Expr          syn typ     :: Type
                       varmap  :: [(Name, Type)]
```

In the presence of multiple synthesized attributes, the preprocessor ensures that the semantic functions combine them in tuples, but in our program we can simply refer to the attributes by name. The attribute declarations of a single datatype can even be distributed over the program. This overcomes problem 2.

The value of each attribute needs to be defined for every constructor of every data type which has the attribute. These definitions of the semantics of the language are known as "semantic rules", and start with keyword **sem**. An example is:

```
sem Expr | Let
  lhs.listing = "let " ++ @decl.listing ++ " in " ++ @body.listing
```

This states that the synthesized *listing* attribute of a *Let* expression can be constructed by combining the *listing* attributes of its *decl* and *body* children and some fixed strings. The @ symbol in this context should be read as "attribute", not to be confused with Haskell "as-patterns". The keyword **lhs** refers to the parent of the children @*decl* and @*body*, i.e., the nameless *Expr* at the **l**eft **h**and **s**ide of the grammar rule. At the left of the = symbol, the attribute to be defined is mentioned (here the @ symbol may be omitted); at the right, any Haskell expression can be given. The example below shows the use of a **case** expression and an auxiliary function *substit*, applied to occurrences of child attributes. Also, it shows how to use the value of leaves (@*name* in the example), and how to group multiple semantic rules under a single **sem** header:

```
sem Expr
  | Var  lhs.listing = @name
  | Lam  lhs.typ     = Type_Arrow (substit @body.varmap @arg.typ) @body.typ
  | App  lhs.typ     = case @func.typ of
                         (Type_Arrow p b) → substit @arg.varmap b
```

The preprocessor collects and orders all definitions into a single algebra, replacing the attribute references by suitable selections from the results of the recursive tree walk on the children. This overcomes problem 3.

To be able to pass information downward during a tree walk, we can define "inherited" attributes. As an example, it can serve to pass an environment (a lookup table that associates variables to types), which can be consulted when we need to determine the type of a variable:

```
attr Expr inh env :: [(Name, Type)]
sem Expr
  | Var lhs.typ = fromJust (lookup @name @lhs.env)
```

The value to use for the inherited attributes can be defined in semantic rules higher up the tree. In the example, *Let* expressions extend the environment which they inherited themselves with the new environment synthesized by the declaration, in order to define the environment to be used in the body:

```
sem Expr
  | Let body.env = @decl.newenv ++ @lhs.env
```

The preprocessor translates inherited attributes into extra parameters for the semantic functions in the algebra. This overcomes problem 4.

In practice, there are many situations where inherited attributes are passed unchanged as inherited attributes for the children. For example, the environment is passed down unchanged at *App* expressions. This can be quite tedious to do:

```
sem Expr
  | App func.env = @lhs.env
        arg.env  = @lhs.env
```

Since the code above is trivial, the preprocessor has a convention that, unless stated otherwise, attributes with the same name are automatically copied. So, the attribute *env* that an *App* expression inherited from its parent, is automatically copied to the children which also inherit an *env*, and the tedious rules above can be omitted. This captures a pattern that is often addressed by introducing a *Reader* monad [24]. Similar automated copying is performed for synthesized attributes, so if they need to be passed unchanged up the tree, this does not need an explicit encoding, nor a *Writer* monad.

It is allowed to declare both an inherited and a synthesized attribute with the same name. In combination with the copying mechanisms, this enables us to silently thread a value through the entire tree, updating it when necessary. Such a pair of attributes can be declared as if it were a single "threaded" attribute. A useful application is to thread an integer value as a source for fresh variable names, incrementing it whenever a fresh name is needed during the tree walk. This captures a pattern for which otherwise a *State* monad would be needed.

The preprocessor automatically generates semantic rules in the standard situations described, and this overcomes problem 5.

3.2 Ruler, a System for Specifying Type Rule Implementations

With the AG language we can describe the part of a compiler related to tree walks concisely and efficiently. However, this does not give us any means of

looking at such an implementation in a more formal setting. Currently a formal description of Haskell, suitable for both the generation of an implementation and use in formal proofs, does not exist. For EH we make a step in that direction with *Ruler*, which allows us to have both an implementation and a type rule presentation with the guarantee that these are mutually consistent.

With *Ruler* we describe type rules in such a way that both a LaTeX rendering and an AG implementation can be generated from such a common type rule description. We demonstrate the use of *Ruler* by showing *Ruler* code for the Hindley-Milner type inferencing of function application *App* (see previous section for this and other names for expression terms). We omit a thorough explanation of the meaning of these fragments, as our purpose here is to demonstrate how we can describe these fragments with one common piece of *Ruler* source text. Also we do not intend to be complete in our description; we point out those parts corresponding to the distinguishing features of the *Ruler* system.

From a single source, to be discussed below, *Ruler* can both generate a LaTeX rendering for human use in technical writing:

$$\frac{\begin{array}{c} v \text{ fresh} \\ \Gamma;\mathcal{C}^k;v \to \sigma^k \vdash^e e_1 : \sigma_a \to \sigma \rightsquigarrow \mathcal{C}_f \\ \Gamma;\mathcal{C}_f;\sigma_a \vdash^e e_2 : \_ \rightsquigarrow \mathcal{C}_a \end{array}}{\Gamma;\mathcal{C}^k;\sigma^k \vdash^e e_1\ e_2 : \mathcal{C}_a\sigma \rightsquigarrow \mathcal{C}_a} \quad (\text{E.APP}_{HM})$$

and its corresponding AG implementation, for further processing by UUAGC:

```
sem Expr
  | App (func.gUniq, loc.uniq1)
                  = mkNewLevUID @lhs.gUniq
        func.knTy = [mkTyVar @uniq1] `mkArrow` @lhs.knTy
        (loc.ty_a_, loc.ty_)
                  = tyArrowArgRes @func.ty
        arg .knTy = @ty_a_
        loc .ty   = @arg.tyVarMp ⊕ @ty_
```

The given rule describes the algorithmic typing of a function application in a standard lambda calculus with the Hindley-Milner type system. The rule involves four judgements: three premises and a conclusion. All judgements but the one involving the freshness of a type variable have the same structure as these all relate various properties of expressions: the conclusion about the function application $e_1\ e_2$, the premises about the function e_1 and argument e_2.

Ruler exploits this commonality by means of the *scheme* of a judgement, which can be thought of as the type of a judgement:

```
scheme expr =
  holes [node e : Expr, inh valGam : ValGam, inh knTy : Ty
        , thread tyVarMp : C, syn ty : Ty]
  judgeuse tex valGam; tyVarMp.inh; knTy ⊢ .."e" e : ty ⇝ tyVarMp.syn
  judgespec valGam; tyVarMp.inh; knTy ⊢ e : ty ⇝ tyVarMp.syn
```

The scheme declaration for expressions *expr* defines a common framework for the judgements of each *expr* term, such as *App* and *Lam* (lambda expression):

- **holes** : names, types and modifiers of placeholders (or *hole*s) for various properties, such as *e* and *valGam*
- **judgeuse tex** (unparsing): LaTeX pretty printing in terms of holes and other symbols, such as $\vdash$ and $\rightsquigarrow$
- **judgespec** (parsing): concrete syntax for specifying a complete judgement.

Modifiers **node**, **inh**, **syn**, and **thread** are required when generating an AG implementation, to be able to turn a rule into an algorithm. The **thread** modifier introduces two holes with suffix .**inh** and .**syn**, corresponding to an AG threaded attribute. For a LaTeX rendering these modifiers are ignored, but additional formatting is required to map identifiers to LaTeX symbols, for example:

$$
\begin{array}{ll}
\mathit{valGam} & \mapsto \Gamma \\
\mathit{ty} & \mapsto \sigma \\
\mathit{knTy} & \mapsto \sigma^k \\
\mathit{tyVarMp}.\mathbf{inh} & \mapsto \mathcal{C}^k
\end{array}
$$

We omit further discussion of lexical issues.

The rule for function application *App* now is defined by judgements introduced with the keyword **judge**:

$$
\begin{array}{l}
\mathbf{rule}\ \mathit{e.app} = \\
\quad \mathbf{judge}\ \mathit{tvarvFresh} \\
\quad \mathbf{judge}\ \mathit{expr} = \mathit{tyVarMp}.\mathbf{inh}; \mathit{tyVarMp}; (v \rightarrow \mathit{knTy}) \\
\qquad\qquad\qquad \vdash \mathit{eFun} : (\mathit{ty.a} \rightarrow \mathit{ty}) \rightsquigarrow \mathit{tyVarMp.fun} \\
\quad \mathbf{judge}\ \mathit{expr} = \mathit{tyVarMp.fun}; \mathit{valGam}; \mathit{ty.a} \\
\qquad\qquad\qquad \vdash \mathit{eArg} : \mathit{ty.a} \rightsquigarrow \mathit{tyVarMp.arg} \\
\quad - \\
\quad \mathbf{judge}\ \mathit{expr} = \mathit{tyVarMp}.\mathbf{inh}; \mathit{valGam}; \mathit{knTy} \\
\qquad\qquad\qquad \vdash (\mathit{eFun}\ \mathit{eArg}) : (\mathit{tyVarMp.arg}\ \mathit{ty}) \rightsquigarrow \mathit{tyVarMp.arg}
\end{array}
$$

For each judgement its scheme is specified (*expr* in the example). The **judgespec** of the corresponding scheme is used to check the concrete syntax and to bind the holes of the judgement to the concrete values specified by the judgement. From this rule definition a LaTeX rendering can straightforwardly be generated.

For the generation of an AG implementation we need information as specified by hole modifiers. In an AG implementation the structure of the tree drives the choice of which rule to apply. One of the holes needs to correspond to a node of such a tree; the modifier **node** specifies which. Other holes correspond to attributes, which have a direction: top-down (inherited, indicated by modifier **inh**) bottom-up (synthesized, indicated by **syn**) or both (indicated by **thread**).

The judgement with scheme *tvarvFresh* is an example of a judgement which does not fit into a tree structure as required by AG: it does not refer to a **node**

hole. For such schemes, called *relation*s, an explicit AG implementation must be provided. We omit further discussion of relations.

Finally, *Ruler* also provides support for incremental language specification, which we discuss in Section 4.

4 Coping with Design Complexity: Grow Stepwise

To cope with the many features of Haskell, EHC is constructed as a sequence of compilers, each of which adds new features. This enables us to experiment with non-standard features. Fig. 3 shows the standard and experimental features currently introduced in each language variant. The sequence is a didactical choice of increasingly complex features; it is not the development history. Every compiler in the sequence can actually be built out of the repository.

Each language variant in the sequence is described as a delta with respect to the previous language. Usually this delta is a pure addition, but other combinations are possible when:

- language features interact
- the overall implementation and individual increments interact: an increment is described in the context of the implementation of preceding variants, whereas such a context must anticipate later changes.

	Haskell	**extensions**
1	λ-calculus, type checking	
2	type inferencing	
3	polymorphism	
4		higher ranked types, existentials
5	data types	
6	kind inferencing	kind signatures
7	records	tuples as records
8	code generation	GRIN
9	class system	
10		extensible records
11	type synonyms	
12		explicit parameter passing for implicit parameters *
13		higher order predicates *
14–19		*reserved for other extensions* *
20	module system	
95	class instance deriving *	
96		exception handling
97	numbers: Integer, Float, Double	
98	IO	
99	the rest for full Haskell *	

Fig. 3. EH language variants (work in progress is marked by an asterisk '*')

Conventional compiler building tools are neither aware of partitioning into increments nor aware of their interaction. We use a separate tool, called *Shuffle*, to take care of such issues. We describe *Shuffle* in the next section.

For each language variant in the sequence, various artefacts are created, such as example programs, a definition of the semantics, an implementation, and documentation. Fig. 4 shows some of these artefacts for some language variants. The first row shows an example program for each language variant. The second row shows a description of part of the semantics of the language variants (the type rule for functional application), by way of the LaTeX rendering of the type

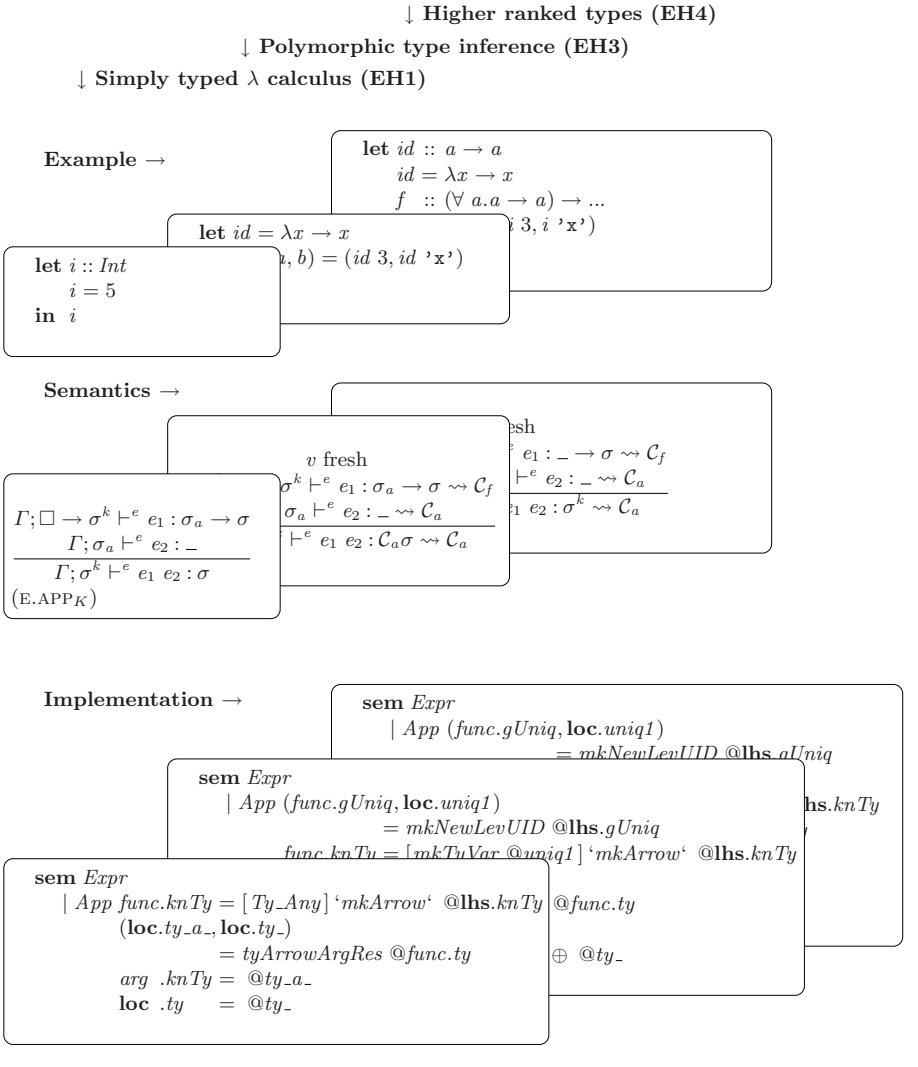


Fig. 4. Examples of created artefacts (rows) for various language variants (columns)

rule generated by *Ruler*. The third row shows the implementation of this type rule in the compiler, by way of the AG output generated by *Ruler* (from the same source). Example language variants shown in the columns of Fig. 4 are EH1 (simply explicitly typed λ-calculus), EH3 (adding polymorphic type inference), and EH4 (adding higher-ranked types).

5 Coping with Maintenance Complexity: Generate, Generate and Generate

For any large programming project the greatest challenge is not to make the first version, but to be able to make subsequent versions. In order to facilitate change, the object of change should be isolated and encapsulated. Although many programming languages support encapsulation, this is not sufficient for the construction of a compiler, because each language feature influences not only various parts of the compiler (parser, structure of abstract syntax tree, type system, code generation, runtime system) but also other artefacts such as specification, documentation, and test suites. Encapsulation of a language feature in a compiler therefore is difficult, if not impossible, to achieve.

We mitigate the above problems by using *Shuffle*, a separate preprocessor. In all source files, we annotate to which language variants the text is relevant. *Shuffle* preprocesses all source files by selecting and reordering those fragments (called *chunk*s) that are needed for a particular language variant. Source code for a particular Haskell module is stored in a single "chunked Haskell" (.chs) file, from which *Shuffle* can generate the Haskell (.hs) file for any desired variant (see Fig. 2, where the stacks of intermediate files denote various variants of a module). Source files can be chunked Haskell code, chunked AG code, but also chunked LaTeX text and code in other languages we use.

Shuffle behaves similar to literate programming tools [27] in that it generates program source code. The key difference is that with the literate programming style program source code is generated out of a file containing program text plus documentation, whereas *Shuffle* combines chunks for different variants from different files into either program source code or documentation.

Shuffle offers a different functionality than version management tools: these offer historical versions, whereas *Shuffle* offers the simultaneous handling of different variants from a single source.

For example, for language variant 2 and 3 (on top of 2) a different wrapper function *mkTyVar* for the construction of the internal representation of a type variable is required. In variant 2, *mkTyVar* is equal to the constructor *Ty_Var*:

$$mkTyVar :: TyVarId \rightarrow Ty$$
$$mkTyVar\ tv = Ty\_Var\ tv$$

However, version 3 introduces polymorphism as a language variant, which requires additional information for a type variable, which defaults to *TyVarCateg_Plain* (we do not further explain this):

$mkTyVar :: TyVarId \rightarrow Ty$
$mkTyVar\ tv = Ty\_Var\ tv\ TyVarCateg\_Plain$

These two Haskell fragments are generated from the following *Shuffle* source:

```
%%[2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv
%%]

%%[3.mkTyVar -2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv TyVarCateg_Plain
%%]
```

The notation `%%[2.mkTyVar` begins a chunk for variant 2 with name *mkTyVar*, ended by `%%]`. The chunk for `3.mkTyVar` explicitly specifies to override `2.mkTyVar` for variant 3. Although the type signature can be factored out, we refrain from doing so for small definitions.

In summary, *Shuffle*:

- uses notation `%%[ ... %%]` to delimit and name text chunks
- names chunks by a variant number and (optional) additional naming
- allows overriding of chunks based on their name
- combines chunks upto an externally specified variant, using an also externally specified variant ordering.

6 Related Work

Compiler building environments. Various compiler construction environments exist [4], originally developed some time ago. We mention Cocktail [22,21], Eli [3,20] and Gentle [5,42]. Of these environments Cocktail provides the most comprehensive and best maintained set of tools for lexical analysis, parsing, attribute grammars and tree transformations. With the exception of Cocktail these environments are focussed upon development in C; Cocktail allows codegeneration for various imperative languages. More specifically for Java, Polyglot [8,34] is an extensible compiler frontend, with many experimental Java extensions.

In contrast, our toolset is built upon Haskell, and wherever possible exploits Haskell's assets such as its strong type system to provide various combinator libraries for (e.g.) parsing instead of traditional generator based solutions. Haskell is also used to specify attribute computations. This allows our tools to be relative lightweight yet comprehensive. It also strongly ties our tools to Haskell, which is benificial in our view because of the compact and descriptive nature of Haskell code fragments.

Tree based systems and Attribute Grammar systems. Computations over abstract syntax trees and tree transformation form a major part of any compiler; Cocktail

also incorporates an attribute grammar system, based on work by Kastens [25], as well as tools for transforming trees. Stratego [45,46] (based on ASF/SDF [2]) is a specialized tool for tree transformations. JastAdd [7,19,23] is an attribute grammar and tree rewrite system built on top of Java, used to build a Java compiler [18].

In contrast, our attribute grammar system is also used for the many tree transformations present in EHC; support for making a modified replica of the tree being analysed makes this work suprisingly well. Occasionally we miss pattern matching features offered by specialized tree transformation tools, but on the other hand we are not limited by the sometimes restricted computational expressiveness offered by these tools.

Declarative specification. Ruler intends to bridge the gap between formal specifications and their implementation, like Tinkertype [30]. Ott [43] provides a typerule based frontend, not for an implementation but for various theorem proving systems. Twelf [6] is a theorem proving system, amongst others used to ultimately specify ML formally in order to proof type safety. Ruler, on the other hand, is currently only used as an implementation tool and documentation tool.

Embedded solutions. All tools mentioned sofar are external tools, in the sense that they generate from a separate specification for a specific implementation language. Monads often have been used for Haskell embedded attribution, e.g. a reader monad corresponds directly to inherited attribution. A similar effect can be accomplished with boilerplate code avoiding approaches [28]. Both approaches work fine for relatively small examples but do not scale well when separate computations have to be merged into one, for example when such computations need each others (intermediate) results.

7 Experiences

Development and debugging. The partitioning into variants is helpful for both development and debugging. It is always clear to which variant code contributes, and if a problem arises one can use a previous variant in order to isolate the problem. Experimentation also benefits because one can pick a suitable variant to build upon, without being hindered by subsequent variants.

However, on the downside, there are builtin systemwide assumptions, for example about how type checking is done. We are currently investigating this issue in the context of *Ruler*.

Use in research and education. EHC is constructed as a library and a toplevel compiler driver (see Fig. 2), facilitating the use of the implementation of EHC by other programs.

We intend to use the first three language variants (Fig. 3) in our basic course on compiler construction, thus providing students with a realistic integrated introduction to language design, compiler implementation, and software engineering. This approach is similar to that in Pierce's textbook [40], however, in

contrast we focus on a realistic implementation of full Haskell instead of small independent implementations of isolated type systems.

Improvements. Although our approach to cope with complexity indeed leads to the advocated benefits, there is room for improvement:

- **Ruler and type rules.** With *Ruler* we generate both AG and LaTeX. *Ruler* notation, AG, and LaTeX have a similar structure. Consequently *Ruler* does not hide as much of the implementation as we would like. We are investigating a more declarative notation for *Ruler*.
- **Loss of information while transforming.** With a transformational approach to different intermediate representations, the relation of later stages to earlier available information becomes unclear. For example, by desugaring to a simpler representation, source code of the user program is reordered and the original source location has to be propagated as part of the AST. Such information flow patterns are not yet automated.
- **High level description and efficiency.** Using a high level description usually also provides opportunities to optimise at a low level. For attribute grammars a large body of optimisations are available [41], some of which are finding their way into our AG system.
- **Stepwise approach vs. aspectwise approach.** EH's stepwise approach imposes a fixed order in which language constructs are implemented on top of each other. Ideally one should be able to arbitrarily combine separate language constructs as aspects (independent implementation fragments), but interaction between language constructs hinders this flexibility. We are investigating the use of aspects in the context of *Ruler*.

Status and plans. We are working towards a release of EHC as a Haskell compiler: variant 99 in the sequence. At the moment, we can compile a prelude and run programs with a bytecode interpreter. We intend to work on AG optimisations, on using LLVM [29] as a backend, and on GRIN global transformations.

References

1. Hugs 98 (2003), `http://www.haskell.org/hugs/`
2. ASF+SDF (2005), `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ASF+SDF`
3. Eli: An Integrated Toolset for Compiler Construction (2005), `http://eli-project.sourceforge.net/`
4. The Catalog of Compiler Construction Tools (2006), `http://catalog.compilertools.net/`
5. The GENTLE Compiler Construction System (2007), `http://gentle.compilertools.net/`
6. The Twelf Project (2007), `http://twelf.plparty.org/wiki`
7. JastAdd (2008), `http://jastadd.org/`
8. Polyglot (2008), `http://www.cs.cornell.edu/projects/polyglot/`

9. Baars, A.: Attribute Grammar System (2004), http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem
10. Bird, R., de Moor, O.: The algebra of programming. Prentice Hall, Englewood Cliffs (1996)
11. Bird, R.S.: Using Circular Programs to Eliminate Multiple Traversals of Data. Acta Informatica 21, 239–250 (1984)
12. Boquist, U.: Code Optimisation Techniques for Lazy Functional Languages, PhD Thesis. Chalmers University of Technology (1999)
13. Boquist, U., Johnsson, T.: The GRIN Project: A Highly Optimising Back End For Lazy Functional Languages. In: Selected papers from the 8th International Workshop on Implementation of Functional Languages (1996)
14. Dijkstra, A.: EHC Web (2004), http://www.cs.uu.nl/groups/ST/Ehc/WebHome
15. Dijkstra, A.: Stepping through Haskell. PhD thesis, Utrecht University, Department of Information and Computing Sciences (2005)
16. Dijkstra, A., Swierstra, S.D.: Typing Haskell with an Attribute Grammar. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 1–72. Springer, Heidelberg (2005)
17. Dijkstra, A., Swierstra, S.D.: Ruler: Programming Type Rules. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 30–46. Springer, Heidelberg (2006)
18. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: OOPSLA, pp. 1–18 (2007)
19. Ekman, T., Hedin, G.: The JastAdd System - modular extensible compiler construction. Science of Computer Programming 69(1-3), 14–26 (2007)
20. Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M.: Eli: a complete, flexible compiler construction system. Communications of the ACM 35(2), 121–130 (1992)
21. Grosch, J., Emmelmann, H.: A Tool Box for Compiler Construction. In: Proceedings of the third international workshop on Compiler compilers (1991)
22. Grosch, J., Vollmer, J.: Compiler Compiler Laboratory (2007), http://www.cocolab.com/
23. Hedin, G., Magnusson, E.: JastAdd, an aspect-oriented compiler construction system. Science of Computer Programming 47(1), 37–58 (2003)
24. Jones, M.P.: Typing Haskell in Haskell. In: Haskell Workshop (1999)
25. Kastens, U.: Ordered Attribute Grammars. Acta Informatica 13, 229–256 (1980)
26. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968)
27. Knuth, D.E.: Literate Programming. Journal of the ACM (42), 97–111 (1984)
28. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical design pattern for generic programming. In: Types In Languages Design And Implementation, pp. 26–37 (2003)
29. Lattner, C.: The LLVM Compiler Infrastructure Project (2007), http://llvm.org/
30. Levin, M.Y., Pierce, B.C.: TinkerType: A Language for Playing with Formal Systems (1999), http://www.cis.upenn.edu/~milevin/tt.html
31. Marlow, S.: The Glasgow Haskell Compiler (2004), http://www.haskell.org/ghc/
32. Marlow, S., Jones, S.P.: The New GHC/Hugs Runtime System (1998), http://citeseer.ist.psu.edu/marlow98new.html

33. Meijer, E., Jeuring, J.: Merging monads and folds for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925. Springer, Heidelberg (1995)
34. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Hedin, G. (ed.) CC 2003 and ETAPS 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
35. Jones, S.P.: Compiling Haskell by program transformation: a report from the trenches. In: European Symposium On Programming, pp. 18–44 (1996)
36. Jones, S.P., Hall, C., Hammond, K., Partain, W., Wadler, P.: The Glasgow Haskell compiler: a technical overview. In: Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference (1992)
37. Jones, S.P., Marlow, S.: Secrets of the Glasgow Haskell Compiler inliner. Journal of Functional Programming, 393–434 (2002)
38. Jones, S.P., Santos, A.: Compilation by Transformation in the Glasgow Haskell Compiler (1994), `http://citeseer.ist.psu.edu/peytonjones94compilation.html`
39. Jones, S.P., Santos, A.: A transformation-based optimiser for Haskell. Science of Computer Programming 32(1-3), 3–47 (1998)
40. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
41. Saraiva, J.: Purely Functional Implementation of Attribute Grammars. PhD thesis, Utrecht University (1999)
42. Schrer, F.W.: The GENTLE Compiler Construction System. R. Oldenbourg Verlag (1997)
43. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., et al.: Ott: effective tool support for the working semanticist. In: ICFP, pp. 1–12 (2007)
44. Doaitse Swierstra, S., Azero Alocer, P.R., Saraiava, J.: Designing and Implementing Combinator Languages. In: Swierstra, D., Henriques, P., Oliveira, J. (eds.) AFP 1998. LNCS, vol. 1608, pp. 150–206. Springer, Heidelberg (1999)
45. Visser, E.: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001)
46. Visser, E.: Stratego Home Page (2005), `http://www.program-transformation.org/Stratego/WebHome`

XHaskell – Adding Regular Expression Types to Haskell

Martin Sulzmann and Kenny Zhuo Ming Lu

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{sulzmann,luzm}@comp.nus.edu.sg

Abstract. We present an extension of Haskell, baptized XHaskell, which combines parametric polymorphism, algebraic data types and type classes found in Haskell with regular expression types, subtyping and regular expression pattern matching found in XDuce. Such an extension proves in particular useful for the type-safe processing of XML data. For example, we can express XQuery and XPath style features via XHaskell combinators. We have implemented the system which can be used in combination with the Glasgow Haskell Compiler.

1 Introduction

Functional programming and XML processing should be a good match. Higher-order functions and parametric polymorphism equip the programmer with powerful abstraction facilities while pattern matching over algebraic data types provides for a convenient notation to specify XML transformations. In the Haskell context, there are a number of tools, for example see [21,29], which provide support for parsing, generating and transforming XML documents.

Unfortunately, XML processing in Haskell does not provide the same static guarantees compared to XML processing in domain specific languages such as XDuce [11] and variants such as CDuce [1]. These languages natively support regular expression types and (semantic) subtype polymorphism [13] and can thus give much stronger static guarantees about the well-formedness of programs. In combination with regular expression pattern matching [12], we can write sophisticated and concise XML transformations.

Previous work attempts to close the gap between XDuce and Haskell but some limitations remain. For example, the work in [3] introduces a pre-processor to provide for regular expression pattern matching. On the down side, the approach is untyped and only supports lists. The combinator library to generate XML values introduced in [27] makes use of the Haskell type class system to check for correctness of constructed values. But neither destruction (pattern matching) nor subtyping among XML values is supported. There are a number of further examples [16,17,18] where Haskell's type extensions are used to encode domain-specific language extensions. While these works are impressive, they often lead to less natural programs compared to writing the same in XDuce.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 75–92, 2008.

In this paper, we introduce an extension of Haskell, baptized XHaskell, which integrates XDuce features such as regular expression types, subtyping and regular expression pattern matching into Haskell. Closely related to our work is XMLambda [19,22]. However, our approach is more powerful because we can express more subtyping relations involving complex types such as $(a^* \mid b^*)$. In addition, we also support the combination of regular expression types and type classes which to the best of our knowledge has not been studied before. We could provide the extension via a combinator library but we chose to write a new front-end (XHaskell type checker and translation scheme to Haskell) which has the advantage that we can support (domain-specific) type error messages and optimizations.

Specifically, our contributions are:

- We introduce XHaskell via examples and demonstrate that the combination of regular expression types with algebraic data types (Section 2), parametric polymorphism (Section 3) and type classes (Section 4) yields a highly expressive system. For example, we can express XQuery and XPath style features via XHaskell combinators.
- We establish sufficient conditions which guarantee that type checking of XHaskell remains decidable (Section 5).
- We have fully implemented the system which can be used in combination with the Glasgow Haskell Compiler. We have taken care to provide meaningful type error messages in case the static checking of programs fails. Our system can possible defer some static checks until run-time (Section 6.1).
- We make use of GHC-as-a-library so that the XHaskell programmer can easily integrate her programs into existing applications and take advantage of the many libraries available in GHC. We also provide a convenient interface to the HaXML parser (Section 6.2).

XHaskell's static semantics is described in terms of a type-directed type-preserving translation from XHaskell to a System F style target language. For brevity, we only give a condensed presentation of the key ideas in Section 5. A complete description is given an accompanying technical report [25]. Further related work in the context of Java, C#, ML and XDuce is discussed in Section 7. Section 8 concludes.

2 Regular Expression and Data Types

In XHaskell we can mix algebraic data types and regular expression types. Thus, we can give a recast of the classic XDuce example also found in [11]. First, we provide some type definitions.

```
data AddressBook = ABook (Person*)
data Person      = Person  Name (Tel?) (Email*)
data Name        = Name String
data Tel         = Tel String
data Email       = Email String
data Entry       = Entry (Name,Tel)
```

The above extends data type definitions as found in Haskell. The novelty is the use of regular expression notation on the right-hand sides. In the definition of `AddressBook` we make use of the Kleene star `*` to describe an address book which consists of an arbitrary sequence of persons. The regular expression operator `?` is used to define optional types such as `Tel?`. Thus, each person is described by a name, an optional telephone number and an arbitrary sequence of email addresses.

In our current implementation we use Haskell's pair syntax to describe XHaskell sequences. We write () to denote the empty sequence and (x, y) to denote sequencing of x and y. Sequences admit more type equality and subtype relations than pairs. For example, sequences are associative, that is $(x, (y, z)) = ((x, y), z)$ and () is the identity among sequences. Sequences subsume pairs and we therefore do not support Haskell style pairs in XHaskell. On the other hand lists and all other data types are still available in XHaskell. The main point of XHaskell is to enrich the Haskell language with additional XDuce features of semantic subtyping and type-based pattern matching as the following examples shows.

Like in Haskell, we can now write functions which pattern match over the above data types. The following function (possibly) turns a single person into a phone book entry.

```
pToE :: Person -> Entry?
pToE (Person (n:: Name) (t::Tel) (es :: Email*)) = Entry (n,t)
pToE (Person (n:: Name) (t::()) (es :: Email*)) = ()
```

The novelty is that we can use a combination of Haskell and XDuce style patterns to define function clauses. For example, consider the first pattern `(Person (n:: Name) (t::Tel) (es :: Email*))`. Like in Haskell, we can pattern match over the constructors of an algebraic data type, here `Person`. In addition, we use XDuce style type-based regular expression patterns to select only a person which has a name, a phone number and an arbitrary number of emails. In the body of the second clause, we use semantic subtyping. The empty sequence value `()` of type `()` is a subtype of `(Entry?)` because the language denoted by `()` is a subset of the language denoted by `(Entry?)`. Hence, we can conclude that the above program is type correct.

The translation scheme for XHaskell's additional features is similar in spirit to the translation of type classes [8]. In target programs, we use a structured representation of values of regular expression types. For example, we use lists to represent sequences and sum types such as `data Or a b = L a | R b` to represent the regular expression choice operator. Thus, the source definition

```
data Person = Person  Name (Tel?) (Email*)
```

translates to the target definition

```
data Person = Person  Name (Or Tel ()) [Email]
```

Some readers may argue why not use the target definition in the first place. That is, use Haskell instead of XHaskell from the start. But then the programmer needs to implement subtyping and regular expression pattern matching herself.

Concretely, in the body of function `pToE` we must insert some explicit tags, here `L` for the first clause and `R` for the second clause, to ensure that the program type checks in Haskell. These tags effectively represent (up-cast) coercions and are automatically inserted by the XHaskell compiler. Similarly, the Haskell programmer must explicitly translate regular expression pattern matching into plain Haskell pattern matching. The XHaskell compiler will automatically insert the (down-cast) coercions, representing the regular expression pattern match, for the programmer. Hence, XHaskell's translation scheme resembles the translation of type classes where specific uses of methods are replaced by concrete type class dictionaries.

To disambiguate the outcome of matching, we employ the longest match policy. For instance, the following program removes the longest sequence of spaces from the beginning of a sequence of spaces and texts.

```
data Space = Space
data Text = Text String

longestMatch :: (Space|Text)* -> (Space|Text)*
longestMatch (s :: Space*, r :: (Space|Text)*) = r
```

The sub-pattern `(s :: Space*)` is potentially ambiguous because it matches an arbitrary number of spaces. However, in XHaskell we follow the longest match policy which enforces that sub-pattern `(s :: Space*)` will consume the longest sequence of spaces. For example, application of `longestMatch` to the value `(Space, Space, Text ``Hello'', Space)` yields `(Text ``Hello'', Space)`.

XHaskell also provides support for XML-style attributes.

```
data Book = Book {{author :: Author?, year :: Year}}
type Author = String
type Year = Int

findBooks :: Year -> Book* -> Book*
findBooks yr (b@Book{{year = yr'}},bs :: Book*) =
   if (yr == yr')
      then (b, findBooks yr bs)
      else (findBooks yr bs)
findBooks yr (bs :: ()) = ()
```

The above program filters out all books published in a specified year. The advantage of attributes `author` and `year` is that we can access the fields within a data type by name rather than by position. For example, the pattern `Book{{year = yr'}}` extracts the year out of a book whereas the pattern `b@` allows one to use `b` to refer to this book.

Attributes in XHaskell resemble labeled data types in Haskell. But there are some differences, therefore, we use a different syntax. The essential difference is that attributes may be optional. For example, `Book {{year = 1997}}` defines an author-less book published in 1997. This is possible because the attribute `author` has the optional type `Author?`. In case of

```
findGoethe :: Book* -> Book*
findGoethe (b@Book{{author = "Goethe", year = _}},bs :: Book*) =
        (b, findGoethe bs)
findGoethe _ = ()
```

the first clause applies if the author is present and the author is Goethe. The pattern `(b@Book{{author = "Goethe", year = _}},bs :: Book*)` could be simplified to `(b@Book{{author = "Goethe"}},bs :: Book*)` because we don't care about the year. In all other cases, i.e. the author is not Goethe, the book does not have an author at all or the sequence of books is empty, the second clause applies. Another (minor) difference between attributes in XHaskell and labeled data types in Haskell is that in XHaskell a attribute name can be used in more than one data type.

```
data MyBook = MyBook {{author :: Author?, year :: Year, price :: Int}}
```

This is more a matter of convenience and relies on the assumption that we use the attribute in a non-polymorphic context only.

To sum up, XHaskell's additional features of regular expression subtyping and pattern matching allow one to write expressive transformations and programs. The XHaskell programs will be more concise and readable compared to writing an equivalent program in Haskell.

3 Regular Expression Types and Parametric Polymorphism

We can also mix parametric polymorphism with regular expressions. Thus, we can write a polymorphic traversal function for sequences similar to the `map` function in Haskell.

```
mapStar :: (a -> b) -> a* -> b*
mapStar f (x :: ()) = x
mapStar f (x :: a, xs :: a*) =  (f x, mapStar f xs)
```

In the above, we assume that type annotations are lexically scoped. For example, variable `a` in the pattern `x::a` refers to `mapStar`'s annotation.

We can now straightforwardly specify a function which turns an address into a phone book by mapping function `pToE` over the sequence of `Persons`.

```
data Book a    = Book a*
type Addrbook  = Book Person
type Phonebook = Book Entry

addrbook :: Addrbook -> Phonebook
addrbook (Book (x :: Person*)) = Book (mapStar pToE x)
```

Notice that the we also support the combination of regular expressions and parametric data types.

Once we have `mapStar` it is not too difficult to define `filterStar` and thus we can express star-comprehension similar to the way list-comprehension are

expressed via `map` and `filter` in Haskell. Star-comprehension provide for a handy notation to write XQuery style programs.

Here is re-formulation of the `findBooks` function using star-comprehension.

```
findBooks' :: Year -> Book* -> Book*
findBooks' yr (bs :: Book*) = [ b | b@Book{{year = yr'}} <- bs, yr == yr']
```

Like list-comprehensions, a star-comprehension consists of a sequence of statements. Concretely, the above star-comprehension has two essential statements. The first statement `b@Book{{year = yr'}} <- bs` is a generator. For each book element `b` in `bs`, we extract the year of publication attribute and bind it to `yr'`. Via the next statement, we then check whether `yr` is equal to `yr'`. If this is the case we return `b`. In XQuery, the above could be written as follows

```
declare function findbooks' ($yr, $bs) {
  for $b in $bs
  where $b/@year = $yr
  return $b
}
```

where the for-clause iterates through a sequence of books, and the where-clause filters out those books were published in year `$yr`.

Parametric polymorphism also poses some challenges. One issue is inference of type instances of polymorphic functions. For example, consider the following `foldStar` function for sequences.

```
foldStar :: (a -> b -> a)-> a -> b* -> a
foldStar f x (y::()) = x
foldStar f x (y::b, ys::b*) =  foldStar f (f x y) ys
```

We infer the missing pattern annotations, which are `f::a->b->a` and `x::a`, using well-established techniques [9,12]. Thus, we can straightforwardly infer that `foldStar` is used at type instance `(a -> b -> a)-> a -> b* -> a` by applying standard local inference methods [20]. Similar methods are also applied in other languages such as GenericJava and C♯ 2.0. What makes things slightly more complicated for us is the presence of subtyping.

Let's consider an example to explain this point in more detail. Suppose we use `foldStar` to build more complex transformations. For example, we want to transform a sequence of alternate occurrences of `a`'s and `b`'s such that all `a`'s occur before the `b`'s. We can specify this transformation via `foldStar` as follows

```
transform :: (a|b)* -> (a*,b*)
transform xs = foldStar ((\x -> \y -> case y of
                           (z::a) -> (z,x)
                           (z::b) -> (x,z)
                         ) :: (a*,b*) -> (a|b) -> (a*,b*))
                () xs
```

We assume that the types of lambda-bound variables are explicitly provided. See the type annotation in the function body. The challenge here is to infer that `foldStar` is used at type instance

```
((a*,b*)->(a|b)->(a*,b*))->(a*,b*)->(a|b)*->(a*,b*)
```

From the types of the arguments and the result type of `transform`'s annotation we infer the type

```
((a*,b*)->(a|b)->(a*,b*))->()->(a|b)*->(a*,b*)
```

But this type does not exactly match the above type. The mismatch occurs at the second argument position. Our solution is to take into account subtyping when checking for type instances. We find that $\vdash$ `()` $\leq (a^*, b^*)$ and therefore the above program is accepted.

A second issue when combining parametric polymorphism and regular expressions is to guarantee that the meaning of programs remains unambiguous. The following function filters out all a's out of sequence of a's or b's.

```
filter :: (a|b)* -> b*
filter (x :: b, xs :: (a|b)*) = (x, filter xs)
filter (x :: a, xs :: (a|b)*) = filter xs
filter () = ()
```

The question is what happens if we use `filter` at type instance `(C|C)* -> C*` where `C` is some arbitrary type? XHaskell functions are type-checked and translated independently from any specific use site. This is clearly important to ensure modularity. The consequence is that we unexpectedly may filter out all `C`'s if we apply `filter` to a sequence of `C`'s. On the other hand, the monomorphized version

```
filterCC :: (C|C)* -> C*
filterCC (x :: C, xs :: (C|C)*) = (x, filterCC xs)
filterCC (x :: C, xs :: (C|C)*) = filterCC xs
filterCC () = ()
```

will not filter out any `C`'s at all. To summarize. The issue is that a polymorphic function used at a monomorphic instance may behave differently compared to the monomorphized function. To be clear, there are no (type) soundness issues. A solution is to reject ambiguous uses of `filter` by checking the instantiation sites. The instance `(C|C)* -> C*` is ambiguous whereas the instance `(A|B)* -> B*` is clearly fine (that is unambiguous). At the moment, we accept potentially ambiguous types. We believe that this is acceptable because our main goal is to strive for expressiveness and soundness without limiting the set of acceptable programs.

4 Regular Expression Types and Type Classes

XHaskell also supports the combination of type classes and regular expression types. For example, we can define `(*)` to be an instance of the `Functor` class.

```
instance Functor (*) where
  fmap = mapStar
```

In our next example we define an instance for equality among a sequence of types.

```
instance Eq a => Eq a* where
  (==) (xs::()) (ys::()) = True
  (==) (x::a, xs::a*) (y::a, ys::a*) = (x==y)&&(xs==ys)
  (==) _ _ = False
```

In our third example, we show how to express a generic set of XPath operations in XHaskell.

```
class XPath a b where
  (//) :: a -> b -> b*

instance XPath a () where
  (//) _ _ = ()

instance XPath a t => XPath a* t where
  (//) xs t = mapStar (\x -> x // t) xs

instance (XPath a t, XPath b t) => XPath (a|b) t where
  (//) (x::a) t = x // t
  (//) (x::b) t = x // t
```

The operation `e1 // e2` extracts all "descendants" of `e1` whose type is equivalent to `e2`'s type.

In our last example, we show that it is very simple to write a pretty-printer for XML data in XHaskell using type classes and regular expression types.

```
class Pretty a where
  pretty :: a -> [Char]

instance Pretty a => Pretty a* where
  pretty xs = foldl (++) [] (mapStar pretty xs)

instance (Pretty a, Pretty b) => Pretty (a|b) where
  pretty (x :: a) = pretty x
  pretty (x :: b) = pretty x

instance (Pretty a, Pretty b) => Pretty (a,b) where
  pretty ((x :: a), (y :: b)) = (pretty x) ++ (pretty y)

instance Pretty () where
  pretty _ = ""

instance Pretty [Char] where
  pretty x = x

instance Pretty Person where
  pretty (Person (n:: Name) (t::Tel?) (es :: Email*)) =
    "<person>" ++ pretty n ++ pretty t ++ pretty es ++ "</person>"
```

```
instance Pretty Name where
  pretty (Name (s :: [Char])) = "<name>" ++ s ++ "</name>"

instance Pretty Tel where
  pretty (Tel (s :: [Char])) = "<tel>" ++ s ++ "</tel>"

instance Pretty Email where
  pretty (Email (s :: [Char])) = "<email>" ++ s ++ "</email>"
```

5 Properties

The meaning of XHaskell is explained via a type-preserving translation scheme to a System F style target language. The translation of programs is driven by the type checking process which boils down to checking subtyping among types. For each pattern we need to check that the pattern type is a subtype of the incoming type. We also need to check that the type of the function body is a subtype of the function's result type.

For concreteness, we give the translation of the earlier `filter` function. See Figure 1. We first list the subtype proof obligations which guarantee that the program is well-typed. The first function clause gives rise to $\vdash (b, (a|b)^*) \leq_{d_1} (a|b)^*$ because of the pattern match and $\vdash (b, b^*) \leq^{u_1} b^*$ because of the function body. The remaining proof obligations resulting from the second and third function clause should be clear.

The idea behind our translation scheme is to extract out of each subtype proof among a proof term (coercion).Specifically, we use up-cast coercions u for the translation of subtyping and down-cast coercions d for the translation of pattern matching among parametric regular expression types. A source expression of type a^* translates to a target expression of type $[a]$ and $(a|b)$ translates to $Or\ a\ b$. [1] Thus, down-cast coercion d_1 emulates the regular expression pattern match in the first clause and up-cast coercion u_3 injects the empty sequence (represented via the unit type in the target program) into the source type b^*. The full details of the translation process are described in [25].

To obtain decidable type checking, we must impose the following two restrictions:

- We only support non-nested data types.
- Subtyping does not extend to type classes.

We explain both points in more detail below.

We say that a data type (definition) is *non-nested* iff

[1] In fact, we use our "own" list type for the translation of the Kleene star. Otherwise, we may possibly encounter overlapping instances in the translated program (though there were none in the source program). For example, the target instance `Pretty [a]` resulting from the source instance `Pretty a*` overlaps with the instance `Pretty [Char]`. We can easily avoid such issues by declaring `newtype XhsList a = XhsList [a]` and use `XhsList a` instead of `[a]`. For convenience, we will stick to standard Haskell lists in the main text.

Source program:

```
filter :: (a|b)* -> b*
filter (x :: b, xs :: (a|b)*) = (x, filter xs)
filter (x :: a, xs :: (a|b)*) = filter xs
filter () = ()
```

Proof obligations resulting from type checking:

1. $\vdash (b, (a|b)^*) \leq_{d_1} (a|b)^*, \vdash (b, b^*) \leq^{u_1} b^*$
2. $\vdash (a, (a|b)^*) \leq_{d_2} (a|b)^*, \vdash b^* \leq^{u_2} b^*$
3. $\vdash () \leq_{d_3} (a|b)^*, \vdash () \leq^{u_3} b^*$

Target program:

```
filter :: [Or a b] -> [b]
filter v =
  case (d1 v) of
    Just (x,xs) = u1 (x, filter xs)
    Nothing ->
      case (d2 v) of
        Just (x,xs) = u2 (filter xs)
        Nothing ->
          case (d3 v) of
            Just () -> u3 ()
            Nothing -> error "non-exhaustive
                              pattern"
```

Up-/Down-cast coercions:

```
d1 :: [Or a b] -> Maybe (b,[Or a b])
d1 [] = Nothing
d1 (x:xs) = case x of
              (R y) -> Just (y,xs)
              _ -> Nothing
```

...

```
u3 :: () -> [b]
u3 () = []
```

Fig. 1. Translation of `filter`

`data T a1 ... an = K t1 ... tm | ...`

and each occurrence of some data type T' in t_i, whose associated declaration `T' a1' ... ak' = ...` is in a strongly connected component with the above declaration, is of the form $T'\ b_1 ... b_k$ where $\{b_1, ..., b_k\} \subseteq \{a_1, ..., a_n\}$. We say a type t is *non-nested* if it is not composed of any nested data types. For example, the non-nested definition

`data T a = Leaf (Maybe [a]) |  Internal (T a) (Maybe Int) (T a)`

is accepted but we reject the nested definition

```
data T2 a = K (T2 [a])
```

Nested definitions are problematic because they may lead to non-termination when checking for subtyping. For example, the subtype proof obligation $\vdash T2\ a \leq T2\ b$ reduces to $\vdash T2\ [a] \leq T2\ [b]$ and so on.

For similar reasons, we impose the restriction that subtyping does not extend to type classes. Recall the declarations

```
class Eq a where
  (==) :: a -> a -> Bool
instance Eq a => Eq a* where ...
```

Suppose some program text gives rise to `Eq (a,a)`. In our subtype proof system, we find that

$$\vdash a^* \to a^* \to Bool \leq^u (a, a) \to (a, a) \to Bool$$

We apply here the co-/contra-variant subtyping rule for functions which leads to $\vdash (a, a) \leq a^*$. The last statement holds. Hence, we can argue that the dictionary E for $Eq\ (a, a)$ can be expressed in terms of the dictionary E' for $Eq\ a^*$ where $E = u\ E'$.

This suggests to refine the type class resolution (also known as context reduction) strategy. Instead of looking for exact matches when resolving type classes with respect to instances, we look for subtype matches. Then, resolution of `Eq (a,a)` with respect to the above instance yields `Eq a`. The trouble is that type class resolution becomes easily non-terminating. For example, `Eq a` resolves to `Eq a` and so on because of $\vdash a \leq a^*$. We have not found (yet) any simple conditions which guarantees termination under a "subtype match" type class resolution strategy. Therefore, we employ a "exact match" type class resolution strategy which in our experience is sufficient. Thus, we can guarantee decidability of type checking.

XHaskell supports type inference in the sense that we exploit local type information, for example provided in the form of user annotations, to infer the type bindings for pattern variables and the type instance at the use site of a polymorphic function. We briefly touched on this issue in Section 3. In XHaskell, we use standard local inference methods [9,12,20]

6 Implementation

We have fully implemented the system as described so far. The XHaskell compiler [30] consists of a type checker and translator. We apply the type-directed translation scheme (sketched in the previous section) and generate Haskell code which compiles under GHC. In the future, we may want to directly compile to GHC's internal GHC's Core language which is a variant of System F. In the following, we discuss a number of topics which concern the practicality of our system.

6.1 Type Error Support

A challenge for any compiler system is to provide meaningful type error messages. This is in particular important in case the expressiveness of the type system increases. The XHaskell compiler is built on top of the Chameleon system [26] and thus we can take advantage of Chameleon's type debugging infrastructure [23,24] to provide concise location and explanation information in case of a type error.

The following program has a type error in the function body because the value `x` of type `(B|A)*` is not a subtype of the return type `(B|C)*`.

```
data A = A
data B = B
data C = C

f :: (B|A)* -> (B|C)*
f (x :: (B|A)*) = x
```

The compiler reports the following error.

```
ERROR: XHaskell Type Error
Expression at:
f (x :: (B|A)*) = x
has an inferred type (B|A)* which is not a subtype of (B|C)*.
Trivial inconsistencies probably arise at:
f :: (B|A)* -> (B|C)*
f (x :: (B|A)*) = x
```

The error report contains two parts. The first part says that a subtyping error is arising from the body of function `f`, namely the expression `x`. The second part points out the cause of the type error. We found literal `A` in `x`'s inferred type, which is not part of the expected type. This is a very simple example but shows that we can provide fairly detailed information about the possible cause of a type error. Instead of highlighting the entire expression we only highlight subexpressions which are involved in the error.

As an extra feature we can post-pone certain type checks till run-time. Let's consider the above program again. The program contains a static type error because the value `x` of type `(B|A)*` is not a subtype of `(B|C)*`. In terms of our translation scheme, we cannot derive the up-cast coercion among the target expression because the subtype proof obligation $\vdash A \leq C$ cannot be satisfied. But if `x` only carries values of type `B*` the subtype relation holds. Hence, there is the option not to immediately issue a static type error here. For each failed subtype proof obligation such $\vdash A \leq C$ we simply generate an "error" case which then yields for our example the following up-cast coercion.

```
u :: [Or B A] -> [Or B C]
u (L b:xs) = (L b):(u xs)
u (R a:xs) = error "run-time failure: A found where B or C is expected"
```

The program type checks now but the translated program will raise a run-time error if the sequence of values passed to function `f` consists of an `A`.

The option of mixing static with dynamic type checking by "fixing" coercions is quite useful in case the programmer provides imprecise type information. In case of imprecise pattern annotations we can apply pattern inference to infer a more precise type. The trouble is that the standard pattern inference strategy [12] may fail to infer a more precise type as shown by the following contrived example.

```
g :: (A,B)|(B,A) -> (A,B)|(B,A)
g (x :: (A|B), y :: (A|B)) = (x,y)
```

It is clear that either (1) `x` holds a value of type `A` and `y` holds a value of type `B`, or (2) `x` holds a `B` and `y` an `A`. Therefore, the above program ought to type check. The problem is that pattern inference computes a type binding for each pattern variable. The best we can do here is to infer the pattern binding $\{(x : (A|B)), (y : (A|B))\}$. But then `(x,y)` in the function body has type `(A|B,A|B)` which is not a subtype of `(A,B)|(B,A)`. Therefore, the above programs fails to type check.

The problem of imprecise pattern inference is well-known [12]. We can offer a solution by mixing static with dynamic type checking. Like in the example above, we generate an up-cast coercion u_2 out of the subtype proof obligation $\vdash (A|B, A|B) \leq^{u_2} (A, B)|(B, A)$ where we use "error" cases to fix failed subtype proofs. This means that application of coercion u_2 potentially leads to a run-time failure (for example, in case we try to coerce (B, B) to $(A, B)|(B, A)$). But the case (B, B) never applies because the incoming types is $(A, B)|(B, A)$. Hence, either case (1) or (2) applies. Hence, for our example there will not be any run-time failure

For the above example, we additionally need to fix the subtype proof $\vdash (A|B, A|B) \leq (A, B)|(B, A)$ resulting from the pattern match check. This check guarantees that the pattern type is a subtype of the incoming type. Out of each such subtype proof we compute a down-cast coercion to perform the pattern match. In case of $\vdash A \leq B$ the pattern match should clearly fail. We can apply the same method for fixing up-cast coercions to also fix down-cast coercions. Each failed subtype proof is simply replaced by an "error" case. The pattern match belonging to the failed subtype proof $\vdash A \leq B$ is fixed by generating

```
\x -> error "run-time failure: we can't pattern match A against B"
```

In our case, we fix $\vdash (A|B, A|B) \leq (A, B)|(B, A)$ by generating

```
d2 :: Or (A,B) (B,A) -> Maybe (Or A B, Or A B)
d2 (L (a,b)) = Just (L a, R b)
d2 (R (b,a)) = Just (R b, L a)
```

Notice that there are no "error" and not even any "Nothing" cases because each of the two components of the incoming type $(A, B)|(B, A)$ fits into the pattern type $(A|B, A|B)$.

6.2 Integration of XHaskell with GHC and HaXML

One of the critical factor for the acceptance of any language extension is the availability of library support and how much of the existing code base can be

re-used. XHaskell makes use of GHC-as-a-library [7] to allow XHaskell programmer to access other Haskell modules/libraries. The XHaskell user simply uses the familiar `import` syntax and the XHaskell compiler will gather all type information from Haskell modules/libraries. We had to extend the existing (typechecking) functionality of GHC-as-a-library to be able to access Haskell type class instances.

Below is an example which shows how to integrate XHaskell with an existing code base.

```
module RSStoXHTML where

import IO          -- Haskell IO module
import RSS         -- RSS XHaskell module generated by dtdToxhs rss.dtd
import XHTML       -- XHTML module generated by dtdToxhs xhtml.dtd
import XConversion -- XHaskell module defining parseXml and writeXml etc

filepath1 = "rss1.xml"
filepath2 = "rss2.xml"

row :: (Link, Title) -> Div
row (Link link, Title title) =
    Div ("RSS Item", B title, "is located at", B link)

filter_rss :: Rss -> Div*
filter_rss rss = [ (row (l,t)) | (Item ( (t :: Title)
                                       , (ts :: (Title|Description)*)
                                       , (l :: Link)
                                       , rs )) <- rss/Channel/Item ]

main :: IO ()
main =  do (rss1 :: Rss) <- parseXml filepath1
           (rss2 :: Rss) <- parseXml filepath2
           let filter_rss1 = filter_rss rss1
               filter_rss2 = filter_rss rss2
           html = Html (Body
                (I ("This document is generated by RSStoXHTML converter,\
                     a program written in XHaskell.")
                , Hr, filter_rss1, filter_rss2))
           writeXml "myrss.xhtml" html
```

Our implementation comes with a tool called `dtdToxhs` which we use here to automatically generate XHaskell datatypes from the RSS and XHTML DTD specifications, for example `RSS`, `Link`, `Title`, `Div` etc. We can then import these data types into our main application. Another XHaskell module `XConversion` provides two functions `parseXml :: String -> IO Rss` to read and validate the RSS (XML) document and `writeXml :: Xhtml -> IO ()` to store the XHTML values into a (XML) file. We read and print from standard I/O. Therefore, we import the Haskell module `IO`. We make use of GHC-as-a-library to

extract type information out of the imported Haskell module `IO`. We use this information to type check and translate the XHaskell program parts.

Function `filter_rss` extracts all `Item` elements out of the RSS document. For each `Item` element we call function `row` to generate an XHTML `Div` element which has the title and the link of this item. We make use of XQuery and XPath-style combinators to extract the immediate child elements of type `Item`. The main function finally generates an XHTML document in which part of the body content is generated using function `filter_rss`. For instance, given the input file `rss1.xml` as follows,

```
<rss>
 <channel>
   <item>
     <title>XHaskell</title>
     <link>http://www.comp.nus.edu.sg/~luzm/xhaskell</link>
   </item>
 </channel>
</rss>
```

and `rss2.xml` as follows,

```
<rss>
 <channel>
   <item>
     <title>Haskell</title>
     <link>http://www.haskell.org/</link>
   </item>
 </channel>
</rss>
```

executing the program RSStoXHTML yields the following XHTML document,

```
<html>
 <body>
  <i>This document is generated by RSStoXHTML converter,
     a program written in XHaskell.</i>
  <hr/>
  <div> RSS Item <b>XHaskell</b>
   is located at <b>http://www.comp.nus.edu.sg/~luzm/xhaskell</b>
  </div>
  <div> RSS Item
   <b>Haskell</b> is located at <b>http://www.haskell.org</b>
  </div>
 </body>
</html>
```

To provide for easier integration of XHaskell with HaXML legacy code, we provide two XHaskell library functions `toHaXml` and `fromHaXml` to convert data from its XHaskell type representation to HaXml type representation and vice versa. Suppose that `haxml_row` is HaXml legacy function which generates a `Div` element out of a `Link` element and a `Title` element. Then we can redefine the function `row` from above as follows.

```
import MyHaXmlLib (haxml_row)
row' :: (Link, Title) -> Div
row' x = fromHaXml (haxml_row (toHaXml x))
```

7 Related Work

In the introduction we have already discussed related work in the context of Haskell. In the context of ML, the work in [4] introduces OCamlDuce which is a merger of OCaml and XDuce. The focus of OCamlDuce is to develop a type inference algorithm to infer types for the OCaml components and most of the XDuce components in a global flow analysis style. The system does not support the combination of parametric polymorphism and regular expression types.

There are a number of works [6,14,15] which extend Java and C# to guarantee type-safety of XML transformations. One of the main aspects of these works is the integration of regular expressions types with the object model in Java and C#. Close to our work is C_ω [2], a language extension of C# to provide first-class support for the manipulation of semi-structured data. C_ω is defined in terms of a type-preserving translation scheme to C# and supports a more limited subtyping relation among semi-structured data compared to our system.

A novel feature of our work is the integration of parametric polymorphism and regular expression. The only prior work we are aware of are is in the context of XDuce [10,28]. Our system can support a richer set of parametric polymorphic types involving regular expressions. See the examples in Section 3. A detailed study of the issues involved in combining parametric polymorphism and regular expressions is beyond the scope of this paper.

The study of improved type error support in the context of regular expression types has only attracted little attention. We are only aware of the work in [5] which proposes a static analysis to check for unused regular expression patterns. This appears to be orthogonal to our type error diagnosis methods. It would be interesting to extend the work in [5] to the combination of regular expressions and data types.

8 Conclusion

We have presented an extension of Haskell which combines parametric polymorphism, algebraic datatype, type class, regular expression types, semantic subtyping and regular expression pattern matching. We have fully implemented the system which can be used in combination of GHC. Our experience so far shows that the system is highly useful in practice. We also provide for an interface to GHC and HaXml to make use of existing libraries and legacy code.

Acknowledgments

We thank Bernd Brassel, Frank Huch and the reviewers for their comments.

References

1. Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-centric general-purpose language. In: Proc. of ICFP 2003, pp. 51–63. ACM Press, New York (2003)
2. Bierman, G., Meijer, E., Schulte, W.: The essence of data access in c_ω. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 287–311. Springer, Heidelberg (2005)
3. Broberg, N., Farre, A., Svenningsson, J.: Regular expression patterns. In: Proc. of ICFP 2004, pp. 67–78. ACM Press, New York (2004)
4. Frisch, A.: OCaml + XDuce. In: Proc. of ICFP 2006, pp. 192–200. ACM Press, New York (2006)
5. Colazzo, D., Castagna, G., Frisch, A.: Error mining for regular expression patterns. In: the 9th Italian Conference On Theoretical Computer Science, pp. 160–172. Springer, Heidelberg (2005)
6. Gapeyev, V., Pierce, B.C.: Regular object types. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 151–175. Springer, Heidelberg (2003); A preliminary version was presented at FOOL 2003
7. Ghc as a library, `http://www.haskell.org/haskellwiki/GHC/As_a_library`
8. Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in Haskell. ACM Transactions on Programming Languages and Systems 18(2), 109–138 (1996)
9. Hosoya, H.: Regular expressions pattern matching: a simpler design (2003)
10. Hosoya, H., Frisch, A., Castagna, G.: Parametric polymorphism for XML. In: Proc. of POPL 2005, pp. 50–62. ACM Press, New York (2005)
11. Hosoya, H., Pierce, B.C.: XDuce: A typed XML processing language (preliminary report). In: Suciu, D., Vossen, G. (eds.) WebDB 2000. LNCS, vol. 1997, pp. 226–244. Springer, Heidelberg (2001)
12. Hosoya, H., Pierce, B.C.: Regular expression pattern matching for XML. In: Proc. of POPL 2001, pp. 67–80. ACM Press, New York (2001)
13. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Trans. Program. Lang. Syst. 27(1), 46–90 (2005)
14. Kempa, M., Linnemann, V.: Type checking in XOBE. In: Proc. Datenbanksysteme fur Business, Technologie und Web, BTW 2003. LNI, pp. 227–246. GI (2003)
15. Kirkegaard, C., Møller, A., Schwartzbach, M.I.: Static analysis of XML transformations in Java. IEEE Transaction on Software Engineering 30(3), 181–0_6 (2004)
16. Kiselyov, O.: HSXML: Typed SXML (2007), `http://okmij.org/ftp/Scheme/xml.html#typed-SXML`
17. Kiselyov, O., Lämmel, R.: Haskell's overlooked object system. Draft; Submitted for journal publication; online since (September 30, 2004); Full version released (September 10, 2005) (2005)
18. Lu, K.Z.M., Sulzmann, M.: An implementation of subtyping among regular expression types. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 57–73. Springer, Heidelberg (2004)
19. Meijer, E., Shields, M.: XMλ: A functional language for constructing and manipulating XML documents (Draft) (1999)
20. Pierce, B.C., Turner, D.N.: Local type inference. ACM Transactions on Programming Languages and Systems 22(1), 1–44 (2000)
21. Schmidt, U.: Haskell XML Toolbox (2007), `http://www.fh-wedel.de/~si/HXmlToolbox`
22. Shields, M., Meijer, E.: Type-indexed rows. In: Proc. of POPL 2001, pp. 261–275. ACM Press, New York (2001)

23. Stuckey, P.J., Sulzmann, M., Wazny, J.: The Chameleon type debugger. In: Proc. of Fifth International Workshop on Automated Debugging (AADEBUG 2003). Computer Research Repository, pp. 247–258 (2003), http://www.acm.org/corr/
24. Stuckey, P.J., Sulzmann, M., Wazny, J.: Type processing by constraint reasoning. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 1–25. Springer, Heidelberg (2006)
25. Sulzmann, M., Lu, K.Z.M.: XHaskell – adding regular expression types to Haskell. Technical report, National University of Singapore (June 2007), http://www.comp.nus.edu.sg/~sulzmann/manuscript/xhaskell-tr.ps
26. Sulzmann, M., Wazny, J.: Chameleon, www.comp.nus.edu.sg/sulzmann/chameleon
27. Thiemann, P.: A typed representation for HTML and XML documents in Haskell. Journal of Functional Programming 12(4-5), 435–468 (2002)
28. Vouillon, J.: Polymorphic regular tree types and patterns. In: Proc. of POPL 2006, pp. 103–114. ACM Press, New York (2006)
29. Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: ICFP 1999, pp. 148–159. ACM Press, New York (1999)
30. Xhaskell, http://xhaskell.googlecode.com

Partial Parsing: Combining Choice with Commitment

Malcolm Wallace

University of York, UK

Abstract. Parser combinators, often monadic, are a venerable and widely-used solution to read data from some external format. However, the capability to return a partial parse has, until now, been largely missing. When only a small portion of the entire data is desired, it has been necessary either to parse the entire input in any case, or to break up the grammar into smaller pieces and move some work outside the world of combinators.

This paper presents a technique for mixing lazy, demand-driven, parsing with strict parsing, all within the same set of combinators. The grammar specification remains complete and unbroken, yet only sufficient input is consumed to satisfy the result demanded. It is built on a *combination* of *applicative* and *monadic* parsers. Monadic parsing alone is insufficient to allow a choice operator to coexist with the early commitment needed for lazy results. Applicative parsing alone can give partial results, but does not permit context-sensitive grammars. But used together, we gain both partiality and a flexible ease of use.

Performance results demonstrate that partial parsing is often faster and more space-efficient than strict parsing, but never worse. The trade-off is that partiality has consequences when dealing with ill-formed input.

1 Introduction

Parser combinators have been with us for a long time. Wadler was the first to notice that parsers could form a monad [12]. Tutorial papers by Hutton and Meijer [5,6] illustrated a sequence of ever-more sophisticated monadic parsers, gradually adding state, error-reporting and other facilities. Röjemo [9] introduced applicative[1] parsers for space-efficiency, whilst Leijen's Parsec [7] aimed for good error messages with both space and time efficiency by reducing the need for backtracking except where explicitly annotated. Packrat parsing [3] eliminates backtracking altogether by memoising results (a technique that is highly space-intensive). Laarhoven's ParseP [11] also eliminates backtracking, by parsing alternative choices in parallel. Swierstra et al have shown us how to do sophisticated error-correction [10], permutation parsing [1], and on-line results through breadth-first parsing [4], all in an applicative style.

But, aside from the latter work, the particular niche of *partial parsing* is still relatively unexplored. A parser, built from almost any of the currently available

[1] The applicative functor is now recognised [8] as a structure simpler than a monad.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 93–110, 2008.

combinator libraries, needs to see the entire input before it can return even a portion of the result. Why is it unusual to be non-strict, demand-driven, partial? Because of the possibility of parse errors. If the document is syntactically incorrect, the usual policy is to report the error and do no onward processing of the parsed data — in order to prevent onward processing, we must wait until all possible errors could have arisen.

Sometimes this is not desirable. Imagine processing a large XML document that is already known to be well-formed. Why should the program wait until the final close-tag has been verified to match its opener, before beginning to produce output? There is also often an enormous memory cost to store the entire representation of the document internally, where lazy processing could in many cases reduce the needed live heap space to a small constant.

Even if we do not know for certain that a document is well-formed, it can still be useful to process an initial part of it. Think too, of an interactive exchange with a user, or a network communications protocol, where input and output must be interleaved.

Of course, there is a flip-side to partial processing – the parsed value may itself be partial, in the sense of containing bottom (undefinedness, or parse errors). One must be prepared to accept the possibility of notification of a parse-failure when it would be too late to undo the processing already completed.

Of all the libraries available, only the one by Hughes and Swierstra [4] has already demonstrated how to achieve partial parsing (they call it 'online' parsing). The framework is applicative in style (rather than monadic) and automatically analyses the grammar to determine when no further errors or backtracking may occur over the part of the input that has already been seen. In the absence of such errors, it becomes possible to return the initial portion of the resultant data structure with confidence that no other parse is possible. (So in fact, their partial values do not contain bottoms.)

However, the mechanism they use to implement this scheme is rather complex, involving polymorphic recursion, and both existential and rank-2 type extensions to Haskell. Whilst undoubtedly powerful, the scheme is also somewhat hard to understand, as witnessed by the fact that no parsing library (except the one which accompanies their paper) has adopted anything like it. The library itself can be fiendishly difficult to modify, even to add simple primitives found in other libraries (e.g. the 'satisfy' of Figure 2).

This paper presents a simpler, more easily understood, method to achieve partial parsing. It avoids scary higher-ranked types, instead continuing to represent parsers in a basic, slightly naive, way. The price to pay is that there is no automated analysis of the parsers, so the decision on where to be lazy or strict is left in the hands of the grammar writer.

We first outline some ordinary (strict) monadic parser combinators, then illustrate how a naive conversion to use a lazy sequencing operator is problematic. An alternative is explored, using a *commit*-based technique to limit backtracking, but this too is found to be inadequate. Finally, it is shown that by mixing

applicative and monadic combinators, the user can gain explicit control over the lazy or strict behaviour of their parsers.

All the combinator variations described here are freely available in the *polyparse* library [14].

1.1 Simple Polymorphic Parsers

An outline of the basic concept and implementation of monadic parsing now follows, with corresponding code in Figure 1. For a fuller treatment, the reader is directed to Hutton and Meijer's comprehensive tutorial [6].

```
newtype Parser t a = P ([t] → (Either String a, [t]))
instance Functor (Parser t) where
  fmap f (P p) = P (λts → case p ts of
                            (Right val, ts') → (Right (f val), ts')
                            (Left msg, ts')  → (Left msg, ts'))
instance Monad (Parser t) where
  return x     = P (λts →   (Right x, ts))
  fail e       = P (λts →   (Left e, ts))
  (P p) >>= q  = P (λts →   case p ts of
                            (Right x,  ts') → let (P q') = q x in q' ts'
                            (Left msg, ts') → (Left msg, ts'))
runParser           :: Parser t a → [t] → (Either String a, [t])
runParser (P p)     = p
onFail              :: Parser t a → Parser t a → Parser t a
(P p) `onFail` (P q) = P (λts → case p ts of
                                  (Left _, _) → q ts
                                  right       → right)
next :: Parser t t
next = P (λts → case ts of
                  [] → (Left "Ran out of input (EOF)", [])
                  (t : ts') → (Right t, ts'))
```

Fig. 1. Basic parser combinators

The *Parser* type is parameterised on the type of input tokens, t, and the type of the result of any given parse, a. A parser is a function from a stream of input tokens, to the desired result paired with the remaining unused tokens. If a parse fails, the failed result is reported in the String alternative of the *Either* type. Many early combinator libraries used *lists* of results to represent multiple ambiguous parses, or failure (if empty). However in practice only the first result is usually of interest, and the empty list unfortunately gives no helpful information in case of errors, hence the design choice here to use the *Either* type.

Parsers are sequenced together using monadic notation, hence the instances of *Functor* and *Monad*. It is clear by inspection of the definition of the sequence operator (>>=), that it is strict in the result of the first parser – it performs a **case** comparison on it.

A parser can be 'run' by applying it to some input token list. The *runParser* function thus lifts embedded parsers out of the monad, back into some calling context.

Choice between different parses is expressed by *onFail*, which tries its second argument parser only if the first one fails. Note that information may be lost, since any error message from the first parser is thrown away. We return to this point later.

Finally, we need a single primitive parser called *next*, that returns the next token in the stream.

Higher-level combinators can be defined using the primitives above. For instance, those in Figure 2.

```
  -- One token satisfying a predicate.
satisfy      :: (t → Bool) → Parser t t
satisfy p    = do {x ← next
                  ; if p x then return x else fail "Parse.satisfy: failed"}
  -- Use 'Maybe' type to indicate optionality.
optional     :: Parser t a → Parser t (Maybe a)
optional p   = (fmap Just p) `onFail` return Nothing
  -- 'exactly n p' parses precisely n items, using the parser p.
exactly      :: Int → Parser t a → Parser t [a]
exactly 0 p  = return []
exactly n p  = do {x ← p
                  ; xs ← exactly (n − 1) p
                  ; return (x : xs)}
  -- Parse a (possibly empty) sequence. Cannot fail.
many         :: Parser t a → Parser t [a]
many p       = do {x ← p
                  ; xs ← many p
                  ; return (x : xs)} `onFail` return []
  -- Parse a sequence followed by a terminator.
manyFinally      :: Parser t a → Parser t z → Parser t [a]
manyFinally p z  = do {xs ← many p
                      ; z
                      ; return xs}
```

Fig. 2. Higher-level combinators built from primitives

A parser for some particular textual data format is then built from these combinators, and looks rather like a recursive-descent grammar. The example in Figure 3 illustrates a grammar for a simplified form of XML. We assume the input tokens have already been lexed according to XML-like rules, and that error messages are easily augmented with positional information. Definitions for less interesting parsers such as *name* and *attribute* are omitted.

```
data Content = Elem String [Attr] [Content]
             | Text String
content   = element `onFail` text
            `onFail` fail "unrecognisable content"
element   = do
  { token "<"
  ; n    ← name
  ; as   ← many attribute
  ; do { token "/>"
       ; return (Elem n as [])}
    `onFail`
    do { token ">"
       ; cs ← manyFinally content (endtag n)
       ; return (Elem n as cs)}
  } `onFail` fail "unrecognisable element"
endtag n = do
  { m ← bracket (token "</") name (token ">")
  ; if n ≡ m then return ()
    else fail ("tag <" ++ n ++ "> terminated by </" ++ m ++ ">")
  }
text      = fmap Text stringToken
            `onFail` fail "unrecognisable text"
token t   = satisfy (≡ t)
```

Fig. 3. Example combinator grammar for a simplified XML

1.2 Problems and Limitations

Complete consumption of input. If we only want a small part of the parsed data, we must still parse the whole thing first. For instance, given the XML input

```
<a><b>hello</b><c>world</c></a>
```

we may wish to extract only the contents of the `<b>` tag, yet are forced to read the `<c>` tag as well! The input could be arbitrarily large, with the fragment of sole interest close to the beginning. Not only that, but the uninteresting part of the input must be fully well-formed, which may be too restrictive for some applications.

One way to avoid complete parsing is to resort to other coding techniques outside the parsing monad. An example of such a technique is repeatedly calling *runParser* on smaller units of the input, tracking unused tokens between calls. Yet manipulation of the parse state is exactly the tedious boilerplate that the monad is supposed to hide! Moving outside the monad also leads to a highly non-modular grammar, requiring much special-case code to deal with the specific fragments of interest.

Ideally, we would like to keep the original grammar, and just interpret it lazily in order to return a partial result.

Error messages are often poor. Due to backtracking over choice points, they rarely point close to the location where the input fails to match the grammar. Indeed, in the worst case, errors are often reported at the topmost outer-most layer of the value's structure, i.e. column 1 of the input.

Using our example XML grammar (Figure 3), the error message from attempting to parse the incorrect input

```
<a><b>hello<b/></a>
```

is not, as one might hope,

```
"tag <b> terminated by </a> at char 18"
```

but rather

```
"unrecognisable content at char 1"
```

Why? Because failure anywhere inside the inner do-blocks of the grammar is thrown away by the enclosing nested *onFail*s, which propagate the failure outwards, but changing the error message at every stage.

One might wonder whether it suffices to re-write *onFail* to preserve and accumulate error messages, rather than ignore them? Unfortunately this only leads to a huge collection of misleading errors, amongst which it is difficult to find the single accurate one.

Backtracking over choices sometimes leads to inefficiency. Again for the example incorrect input

```
<a><b>hello<b/></a>
```

despite the fact that we have already found a valid open tag `<a>` for the element branch of the grammar, nevertheless because something further inside the element is incorrect, this parser necessarily backtracks to the top-level *content* parser and attempts to match the non-element case *text*, on which it is bound to fail.

The XML example only allows for two choices of outer construct – element or text, corresponding to the two branches of the resultant Haskell sum type – but imagine a type and its grammar having a hundred possible different constructors. A parse failure deep within the first branch could lead to the evaluation of all of the remaining 99 constructor choices, failing on all of them, before giving up. Not only is the error message imprecise, but it took much longer than necessary to deliver it!

1.3 Roadmap

In the following sections, we address some of these limitations of the basic parser combinators. First, we make a naive attempt at a lazy parsing monad, to illustrate the conflict between committing to return a value, yet retaining choice. Then we examine whether the prevention of backtracking (second and third

issues above) can not only give better error messages, but also allow a more precise determination of commitment points, at which partial values can be safely returned. Finally, we give a full yet simple solution in which lazy and strict sequencing can be freely mixed.

2 Naive Lazy Monadic Sequencing

It is readily observed that the parser type presented in Figure 1 can either return an error message, or a polymorphic value, but not both. But for partial parsing, we want the parser to return the polymorphic value regardless. Any error due to parse failure could be hidden within the value as an exception, to be triggered only when the immediate subcomponent containing the error is demanded.

Thus, a naive implementation of a lazy monad (corresponding to the strict one already given) is to simply erase the *Either* type constructor, and all *Left* and *Right* value constructors. Any constructions that previously built a *Left* will instead throw an exception. Case branches that previously scrutinised a *Left* can be omitted, and those that scrutinise a *Right* now see the contained value directly – see Figure 4. Furthermore, the sequence operator ($\gg=$) is made lazy by scrutinising the result of its first operand with a (non-strict) **let**-binding, rather than with a **case** as before (the latter would be strict in the tuple pattern).

Sadly though, this approach leaves us with no way to code the choice operator. As the very name *onFail* suggests, the combinator must be able to detect a failure in its left argument before it can try its right argument. But the naive partial parser type no longer represents failure explicitly as a value. Instead, it is a control-flow construct – an exception. One might wonder whether the exception can be caught and handled within the *onFail* combinator, but sadly, we are in the wrong monad! Exceptions can be caught only from the I/O monad, not the parsing monad.

```
newtype Parser t a = P ([t] → (a, [t]))
runParser          :: Parser t a → [t] → (a, [t])
runParser (P p) = p
instance Functor (Parser t) where
  fmap f (P p)  = P (λts → case p ts of
                             (val, ts') → (f val, ts'))
instance Monad (Parser t) where
  return x      = P (λts → (x, ts))
  fail e        = P (λts → (throwException e, ts))
  (P p) >>= q   = P (λts → let (x, ts') = p ts
                               (P q')   = q x
                           in  q' ts')
throwException  :: String → a   -- throw to enclosing I/O monad
```

Fig. 4. A futile attempt at a lazy parsing monad

The lesson here is that the early commitment implicit in returning a partial value, prevents a later choice. So let us examine a different approach, where commitment is made explicit. By annotating the precise locations in the grammar where commitment is possible, it will remain possible to implement choice everywhere else.

3 Choice and Commitment

The introduction of explicit commitment is initially motivated by a desire to improve error reporting. We have already seen how backtracking over choice points leads to poor error messages. But in addressing this problem, we will disallow backtracking at defined locations, and therefore also eliminate choice there too. The hope is that this will enable us to return a partial result at that same location.

Essentially, parse failures can be divided into two separate classes: recoverable and unrecoverable. Recoverable errors allow backtracking through any enclosing choice point; unrecoverable errors should always be reported to the user – they override any enclosing choice point.

We refine the original parser type to codify the different error classes – see Figure 5. Instead of the plain *Either* type, we introduce *Result*, which gives a three-valued logic: success, failure, or a committed result. The committed result is the mechanism used to prevent backtracking. Ultimately there is of course no semantic difference between a plain success or a committed success. But a commitment that ends up being a failure cannot be recovered – it must be reported. By contrast, the choice combinator can throw away an uncommitted failure, to try some other branch.

Figure 5 shows how the basic monadic machinery is modified for this new representation. The choice combinator tries alternatives only when errors are recoverable – after commitment, no alternative is possible, just as surely as if the result of the first operand were successful.

Finally, we add the new combinator *commit*, which serves as the primary mechanism for a grammar-writer to indicate where sufficient tokens have been seen to be certain that no alternative parse path is possible.

Commit is a kind of dual of the *try* combinator in Parsec [7]. In Parsec, no backtracking is allowed normally – it must be explicitly permitted with *try*. But in our framework, backtracking is normally the default, except where explicitly disallowed by *commit*. Ultimately, they have a similar effect however: the calling context of *try* or *commit* will never be returned to; in both cases, we have committed to any particular branch that led to the current call, yet are still willing to try different alternative branches inside the argument to *commit*.

Commit is similar to the *cut* operator used by Röjemo's combinators [9] to achieve space efficiency. Indeed, it solves the very same space-leak, which is also identified by Leijen as a primary motivator for developing Parsec [7]. *Commit* also bears a strong similarity to the extra-logical ! (cut) operator in Prolog, which serves to prevent backtracking in its implementation model.

```
data Result t a  = Success a [t]
                 | Failure String [t]
                 | Commit (Result t a)
newtype Parser t a = P ([t] → Result t a)

runParser        :: Parser t a → [t] → (Either String a, [t])
runParser (P p) = result ∘ p
  where
    result (Success a ts) = (Right a, ts)
    result (Failure e ts) = (Left e,  ts)
    result (Commit r)     = result r

instance Monad (Parser t) where
  return x     = P (Success x)
  fail e       = P (Failure e)
  (P p) >>= q = P (continue ∘ p)
    where continue (Success x ts) = let (P q') = q x in q' ts
          continue (Failure e ts) = Failure e ts
          continue (Commit r)     = Commit (continue r)

onFail                 :: Parser t a → Parser t a → Parser t a
(P p) `onFail` (P q) = P (λts → case p ts of
                                  (Failure _ _) → q ts
                                  r             → r)

commit                 :: Parser t a → Parser t a
commit (P p)           = P (Commit ∘ p)
```

Fig. 5. Parsers with commitment, for better error-reporting

Figure 6 refines the example XML grammar of Figure 3, re-expressing it in terms of *commit*. Note the careful placement of commitment after sufficient tokens have been read to disambiguate the cases. Now, when given the badly-formed input string

```
<a><b>hello<b/></a>
```

in contrast to the previous attempt, we receive the error message

```
"<b> terminated by </a> at char 18"
```

as hoped.[2] The *endTag* parser is responsible for generating the message, and the nearest enclosing *commit* (in the second branch of *element*) is responsible for ensuring that it (and no other message) is reported.

It is worth noting that one of the commonest sources of bugs in Parsec grammars is that users do not know where to place the *try* combinator. Parsec grammars are *LL(1)* by default, but *try* is used to permit extra lookahead for disambiguation. It can be difficult to look at a grammar and count the required lookahead. This leads to the curious observation that Parsec grammars are not

[2] Different implementations of the *manyFinally* combinator can yield even more detailed error messages.

```
element = do
  { token "<"
  ; commit (do
      { n    ← name
      ; as   ← many attribute
      ; do { token "/>"
           ; commit (return (Elem n as []))}
        `onFail`
        do { token ">"
           ; commit (do { cs ← manyFinally content (endtag n)
                        ; return (Elem n as cs)})}
      } `onFail` fail "unrecognisable element")
  }
```

Fig. 6. The XML grammar for 'element', re-expressed using *commit*. (Other productions remain unchanged.)

in fact compositional! When a user plugs two previously-working grammars together, the combination often turns out not to work as expected, and they resort to simply sprinkling *try* into various locations to discover a fix.

By contrast, we believe that the *commit* approach is superior, because the lack of a *commit* will not cause the grammar to fail unexpectedly, merely to be inefficient or to give unhelpful error messages. In addition, the intuition needed to place a *commit* combinator correctly within the grammar is a much lower barrier. It indicates a simple certainty that no alternative parse is possible once this marked point has been reached. This is easier to verify by inspection than deciding how many tokens of lookahead are required to disambiguate alternatives.

4 How to Be Lazy

Does the form of explicit commitment described above help to achieve partial results? Sadly the answer is no, at least not directly. Once the parser has emitted a *Commit* constructor, it has still not determined whether the result will be a success or failure. And even if it does turn out to be a success, we do not know (at the moment of commitment) which constructor of the successful polymorphic value is going to be returned. Indeed, there is no way to discover it, *because* the result of *commit* is fully polymorphic – by definition the combinator cannot know anything about the enclosed value's representation.

Thus, the insight gained is that we need a combinator which, in addition to explicitly marking the point of commitment to a value, must know enough about that value to return a portion of it immediately. Commitment must be parameterised on the thing we are committing to.

Furthermore, some new form of sequencing combinator is required, which can build a whole value from component parts, but is capable of returning a partially composed value before the end of the sequence is complete. For this, we must leave behind the monadic world, especially monadic sequence. Some

strict sequencing will remain useful, but short of composing multiple monads, we cannot mix lazy and strict sequences using only the monadic framework.

It turns out that the world of *applicative functors* [8] is a more convenient place to find the kind of sequence we want. In particular, functorial *apply* can be viewed as a sequencing operator. The correspondence to monadic bind (and the difference) is clearest when the arguments to *apply* are flipped:

$$\begin{array}{ll} \mathit{apply} & :: \mathit{Applicative}\ f \Rightarrow f\ (a \to b) \to f\ a \to f\ b \\ \mathit{flip\ apply} & :: \mathit{Applicative}\ f \Rightarrow f\ a \to f\ (a \to b) \to f\ b \\ (\gg\!=) & :: \mathit{Monad}\ m \Rightarrow m\ a \to (a \to m\ b) \to m\ b \end{array}$$

Some existing parser combinator libraries are based on applicative functors, rather than monads [4,9]. *Apply* is less powerful than monadic bind, in the sense that the former can be implemented in terms of the latter, but not vice versa. This captures the intuition that *apply* simply combines functorial values, that is, the order of evaluation of left and right arguments is not restricted, so one cannot depend on the other. By contrast, the monadic bind allows the contents of the functorial value to be examined, named, and used, in the sequel. Thus, the monadic style allows context-sensitive parsing, whilst the applicative style is context-free.

There is a straightforward and obvious definition of *apply* in terms of bind:

$$pf\ \text{`}\mathit{apply}\text{`}\ pa = \mathbf{do}\ \{f \leftarrow pf; a \leftarrow pa; \mathit{return}\ (f\ a)\}$$

but of course this is no good for returning partial results, because as we already know, the monadic bind is insufficiently partial – that is the problem we are trying to overcome. Instead, we can define *apply* to always succeed and return a result, if its left argument succeeds. For instance, if the value delivered by the left functorial argument is a partially-applied data constructor, and the right argument delivers the next component of that constructor, then we can immediately return the constructor portion of the value, before we know whether the component to be contained within it is fully parse-correct.

In the formulation of Figure 7, we revert to the original *Either* variant of the *Parser* datatype, but could equally have used the *Result* variant associated with the *commit* combinator. The improved error-reporting of the latter is entirely independent of, and orthogonal to, the issue of partiality. A point of special note is that the use of the *Either* type for parsing continues to allow the original implementation of the choice combinator *onFail*.

But the key point in this definition of *apply* is that if the first parser succeeds, then the whole combined parse succeeds (returns a *Right* value). Both failures and successes within the second parser are stripped of their enclosing *Left* or *Right*, and used 'naked'. The new *runParser* is the place where the *Either* wrapper is discarded, leaving just the naked value (or exception).

For illustration, Figure 8 re-expresses the XML grammar once again, this time in a lazy fashion. Application is of course curried, so chaining many parsers together is as straightforward in the applicative case as in the monadic case. Note how a mixture of strict monadic sequence and lazy application is used, and

```
newtype Parser t a = P ([t] → (Either String a, [t]))
runParser          :: Parser t a → [t] → (a, [t])
runParser (P p) = convert ∘ p
  where convert (Right a, ts) = (a, ts)
        convert (Left e, ts)  = (throwException e, ts)
infixl 3 `apply`
apply              :: Parser t (a → b) → Parser t a → Parser t b
(P pf) `apply` pa = P (continue ∘ pf)
  where
    continue (Left e,  ts) =  (Left e, ts)
    continue (Right f, ts) =  let (a, ts') = runParser pa ts
                              in  (Right (f a), ts')
```

Fig. 7. A parser that mixes monads and applicative functors. (The instances of Monad and Functor classes, and the implementation of *onFail* remain exactly as in Figure 1.)

```
element = do
  { token "<"
  ; return Elem
      `apply` name
      `apply` many attribute
      `apply` (do { token "/>"
                  ; return [] }
               `onFail`
               do { token ">"
                  ; manyFinally content (endtag n) })
  } `onFail` fail "unrecognisable element"
```

Fig. 8. The XML 'element' grammar in lazy form

how easily strict sequence (with the ability to backtrack over choices) sits inside an enclosing applicative (partial, lazy) sequence.

It is also worth making the point that this revised grammar no longer checks that XML end tags match their opening tags *in advance* of returning the prefix of the element. The check will only occur once the final inner content is demanded by the context of the parser.

So, now that we have two ways to express sequence with combinators, the user must develop their grammar to make careful use of lazy or strict sequence as appropriate. Many of the non-basic combinators must be checked carefully to ensure that they are sufficiently lazy. For example, if we want *exactly* (from Figure 2) to return a lazy list without waiting for all elements to become available, we must rewrite the earlier definition as follows:

```
exactly       :: Int → Parser t a → Parser t [a]
exactly 0 p  = return []
exactly n p = do x ← p
                 return (x:) `apply` exactly (n − 1) p
```

5 Evaluation

5.1 Performance

To give a flavour of the performance of lazy partial parsing, we designed a small number of (slightly artificial) tests using the Xtract tool from the HaXml suite [13,15]. Xtract is a grep-like utility which searches for and returns fragments of an XML document, given an XPath-like query string. Because the intention is to find small parts of a larger document, it is an ideal test case for partial parsing. The XML parser used by Xtract is switchable between the strict and lazy variations[3].

We created a number of well-formed XML documents of different sizes n (ranging on a logarithmic scale from 10 to 1,000,000) with interesting characteristics:

- linear: the document is a flat sequence of n identical elements enclosed in a single wrapper element.
  ```
  <file> <element/> <element/> ... </file>
  ```
- nested: the document contains n elements of different types, with element type i containing a single element of type $i + 1$ nested inside it, except for the nth element, which is empty.
  ```
  <file> <element0>
           <element1>
             <element2> ...
             </element2>
           </element1>
         </element0>
  </file>
  ```
- follow: the nested document, followed by a single trivial element, together enclosed in a wrapper element.
  ```
  <file> <element0>
           <element1> ...
         </element0>
         <follow/>
  </file>
  ```

The queries of interest are:

- `Xtract "/file/element[0]" linear`
  Find the first element in the flat sequence of elements.
- `Xtract "/file/element[$]" linear`
  Find the last element in the flat sequence of elements.
- `Xtract "//element`n`" nested`
  Find the most deeply nested element(s) in the nesting hierarchy. The difference between this test and the following one is that this test continues searching after finding the first result.

[3] Software releases HaXml-1.20 and polyparse-1.2 together contain all the test code.

- `Xtract "//element`n`[0]" nested`
  Find only the first most deeply nested element in the nesting hierarchy.
- `Xtract "/file/follow" follow`
  Find the single top-level element that follows the large deeply-nested element.

The time and memory taken to satisfy each query is given in Tables 1 and 2, using both the strict and lazy parser variations. In all cases, the lazy parser is better (both faster, and more space efficient) than the strict parser. For extremely large documents, where the strict parser often crashes due to stack overflow, the lazy parser continues to work smoothly. For the cases where the only result is a small, early, fragment of the full document, laziness reduces the complexity of the task from linear to constant, that is, it depends on the required distance into the document, not on the size of the document. Even when the searched element is at the end of the linear document, the lazy version is orders of magnitude faster, for large inputs.

The difference between the resources used by the lazy queries for the first vs. all nested elements is interesting. Taking the first element is almost exactly twice as fast, and half as space-hungry, as looking for all elements. This corresponds exactly to the intuition that the latter needs to check all closing tags against their openers (of which there are equal numbers), whilst the former only needs to look at the opening tags.

None of this is very surprising of course. Lazy streaming is well-known to improve the complexity of many algorithms operating over large datasets, often allowing them to scale to extreme sizes without exhausting memory resources, where a more strict approach hits physical limitations. One such demonstration is given in the field of isosurface extraction for visualisation [2], where the pure lazy solution in Haskell is slower than a rival C++ implementation, only until very large inputs are considered, beyond which the Haskell overtakes the C++.

5.2 Comparisons

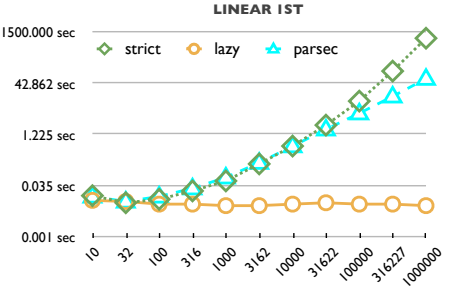
How does lazy parsing fare against other combinator libraries? Parsec claims to be "industrial-strength" and very fast. In contrast, the combinators presented here are somewhat simplistic, with no particular tuning for speed. So for comparison, we reimplemented our XML parser using Parsec: some selected measurements are incorporated in Table 1. Indeed, Parsec is in general faster than our strict library, but slower than our lazy library. Depending on the nature of the test, Parsec's performance aligns pretty closely to either the strict or lazy variations. Nevertheless, laziness always wins hugely when it is able to reduce a linear search to constant time.

The Utrecht combinators claim to be both partial and even faster than Parsec, so we also attempted to reimplement our XML parser in this framework too, to take advantage of the laziness. Unfortunately, the Utrecht library is entirely applicative in nature. Thus, it was not possible to implement the context-sensitive monadic parser needed for XML. (The accompanying paper [4] does give an illustrative instance of monad, but the real implementation of the library is so far

Table 1. Time performance results, measured on a twin-core 2.3GHz PowerPC G5, with 2Gb physical RAM. All timings are best-of-three, measured in seconds by the unix time command (user+system). The graph plots use a log-log scale. Blank entries indicate stack overflow.

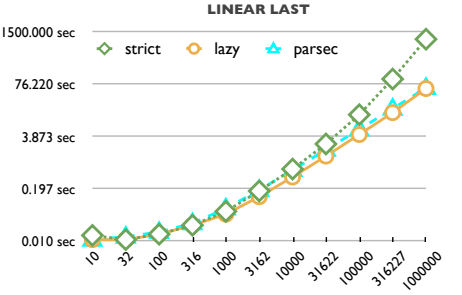
LINEAR 1ST

N	STRICT	LAZY	PARSEC
10	0.018	0.013	0.017
32	0.011	0.012	0.012
100	0.014	0.01	0.017
316	0.025	0.01	0.03
1000	0.05	0.009	0.064
3162	0.163	0.009	0.177
10000	0.563	0.01	0.558
31622	2.407	0.011	1.795
100000	12.733	0.01	5.471
316227	102.272	0.01	17.685
1000000	1001.411	0.009	60.321



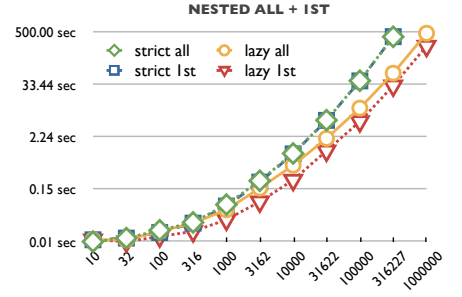
LINEAR LAST

N	STRICT	LAZY	PARSEC
10	0.014	0.011	0.011
32	0.011	0.011	0.013
100	0.015	0.015	0.017
316	0.025	0.025	0.028
1000	0.055	0.047	0.065
3162	0.176	0.127	0.186
10000	0.614	0.39	0.579
31622	2.565	1.285	1.853
100000	13.594	4.395	5.907
316227	103.752	15.299	19.182
1000000	1005.715	60.389	62.645



NESTED ALL + 1ST

N	STRICT ALL	LAZY ALL	STRICT 1ST	LAZY 1ST
10	0.01	0.011	0.011	0.011
32	0.012	0.011	0.012	0.01
100	0.018	0.017	0.016	0.013
316	0.027	0.026	0.026	0.017
1000	0.068	0.052	0.063	0.031
3162	0.233	0.159	0.218	0.079
10000	0.957	0.52	0.937	0.242
31622	5.348	2.088	5.303	1.097
100000	41.25	10.001	40.839	5.012
316227	400.36	60.854	403.932	31.86
1000000		480.729		247.842



FOLLOW

N	STRICT	LAZY	PARSEC
10	0.01	0.01	0.011
32	0.012	0.011	0.011
100	0.016	0.013	0.016
316	0.027	0.022	0.027
1000	0.061	0.045	0.066
3162	0.219	0.144	0.236
10000	0.907	0.541	0.958
31622	5.043	2.735	4.787
100000	40.813	18.354	34.251
316227	400.88	158.205	302.285
1000000		1501.799	2845.978

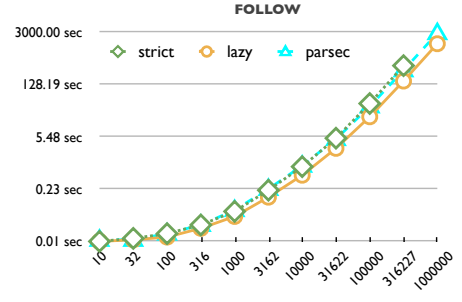
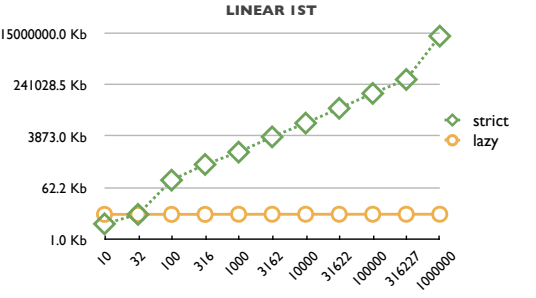


Table 2. Memory performance results. All measurements are of peak live heap usage, measured in kilobytes. The graph plots use a log-log scale.

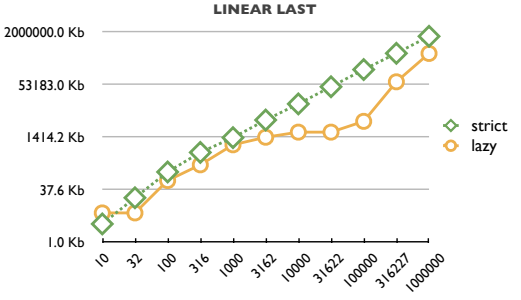
LINEAR IST

N	STRICT	LAZY
10	3.6	7.7
32	7.7	7.7
100	120	7.7
316	423	7.7
1000	1137	7.8
3162	3872	7.8
10000	11664	7.8
31622	38360	7.8
100000	126535	7.8
316227	386172	7.8
1000000	12539803	7.8



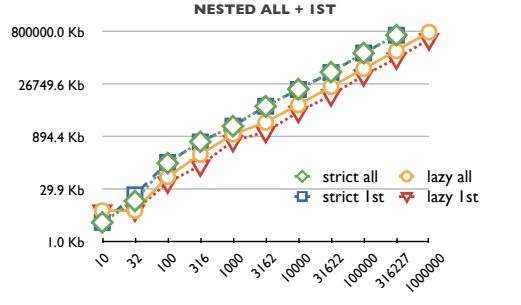
LINEAR LAST

N	STRICT	LAZY
10	3.6	7.7
32	21.9	7.7
100	130	72
316	501	213
1000	1389	848
3162	4689	1430
10000	14266	2032
31622	46595	2032
100000	152593	4284
316227	468579	65557
1000000	1514577	464335



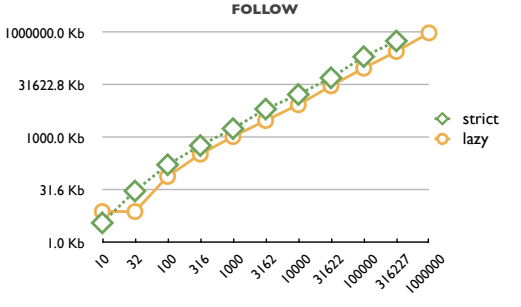
NESTED ALL + IST

N	STRICT ALL	LAZY ALL	STRICT IST	LAZY IST
10	3.6	7.7	3.6	7.7
32	14.4	7.7	21.4	7.7
100	166	69	174	47.4
316	662	291	653	142
1000	1812	1077	1818	694
3162	6562	2324	6567	1250
10000	19876	7193	19445	4562
31622	59790	23181	57809	13293
100000	204901	74874	205069	48605
316227	654928	235890	650622	143416
1000000		792254		498492



FOLLOW

N	STRICT	LAZY
10	3.7	7.8
32	30	7.8
100	169	78.2
316	596	338
1000	1811	1074
3162	6562	3082
10000	17112	8713
31622	51169	30134
100000	204901	96290
316227	577197	288569
1000000		979541



removed from the paper's simplified presentation that it proved too difficult to translate.)

6 Conclusion

The main contribution of this paper is a demonstration that partial parsing is both possible, and convenient, using a framework with a mixture of monadic and applicative parser combinators. Applicative sequence is used for lazy sequencing, and monadic bind for strict sequence.

The decision on where a grammar should be strict and where lazy, is left to the programmer. This differs from the only other extant library to deliver partial parsing [4], which can automatically analyse the grammar to determine where laziness is possible.

As expected, the resources needed to partially parse a document depend on how much of the input document is consumed, not on the total size of the document. Nevertheless, if the whole document is demanded, it is still cheaper to parse it lazily than strictly.

However, partial parsing also means that the ability to report parse errors is shifted from within the parsing framework out to the world of exception handling.

A secondary contribution is the re-discovery of the *commit* combinator to prevent backtracking and enable both better error-reporting and space-efficiency. Although *commit* was previously known [9] to remove a particular space leak associated with choice, the impact on error-reporting was not so widely appreciated. Parsec's *try*, as a dual to *commit*, is more commonly used for this purpose, but is rather less useful due to the need for a correct manual analysis of the grammar for lookahead, and the difficulty of doing this. By contrast, placement of *commit* is not required for correctness, only for efficiency, and the manual analysis involved is easy.

References

1. Baars, A., Löh, A., Swierstra, D.: Parsing permutation phrases. Journal of Functional Programming 14(6), 635–646 (2004)
2. Duke, D., Wallace, M., Borgo, R., Runciman, C.: Fine-grained visualization pipelines and lazy functional languages. IEEE Transactions on Visualization and Computer Graphics 12(5), 973–980 (2006)
3. Ford, B.: Packrat parsing: Simple, powerful, lazy, linear time. In: International Conference on Functional Programming, Pittsburgh, October 2002. ACM SIGPLAN, New York (2002)
4. Hughes, R.J.M., Swierstra, S.D.: Polish parsers, step by step. In: Proceedings of ICFP, Uppsala, pp. 239–248. ACM Press, New York (2003)
5. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming 2(3), 323–343 (1992)
6. Hutton, G., Meijer, E.: Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham (1996)
7. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, University of Utrecht (2001)

8. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 17(5), 1–13 (2007)
9. Röjemo, N.: Garbage collection and memory efficiency in lazy functional languages. PhD thesis, Chalmers University of Technology (1995)
10. Swierstra, D.: Combinator Parsers: from toys to tools. ENTCS, vol. 41. Elsevier, Amsterdam (2001)
11. van Laarhoven, T.: ParseP: software distribution, `http://twan.home.fmf.nl/parsep/`
12. Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Marktoberdorf Summer School on Program Design Calculi, August 1992. NATO ASI Series F: Computer and systems sciences, vol. 118. Springer, Heidelberg (1992)
13. Wallace, M.: HaXml software distribution, `http://haskell.org/HaXml`
14. Wallace, M.: Polyparse combinators (2007), `http://www.cs.york.ac.uk/fp/polyparse`
15. Wallace, M., Runciman, C.: Haskell and XML: generic combinators or type-based translation? In: Proceedings of ICFP, Paris. ACM Press, New York (1999)

Lazy Contract Checking for Immutable Data Structures

Robert Bruce Findler, Shu-yu Guo, and Anne Rogers

University of Chicago
{robby,arc,amr}@cs.uchicago.edu

Abstract. Existing contract checkers for data structures force programmers to choose between poor alternatives. Contracts are either built into the functions that construct the data structure, meaning that each object can only be used with a single contract and that a data structure with an invariant cannot be viewed as a subtype of the data structure without the invariant (thus inhibiting abstraction) or contracts are checked eagerly when an operation on the data structure is invoked, meaning that many redundant checks are performed, potentially even changing the program's asymptotic complexity.

We explore the idea of adding a small, controlled amount of laziness to contract checkers so that the contracts on a data structure are only checked as the program inspects the data structure. Unlike contracts on the constructors, our lazy contracts allow subtyping and thus preserve the potential for abstraction. Unlike eagerly-checked contracts, our contracts do not affect the asymptotic behavior of the program.

This paper presents our implementation of these ideas, an optimization in our implementation, performance measurements, and a discussion of an extension to our implementation that admits more expressive contracts by loosening the strict asymptotic guarantees and only preserving the amortized asymptotic complexity.

1 Introduction

Assertion-based contracts play an important role in constructing robust software. They give programmers a technique to express program invariants in a familiar notation with familiar semantics. Contracts are expressed as program expressions of type boolean. When the expression's value is true, the contract holds and the program continues. When the expression's value is false, the contract fails, causing the contract checker to abort the program, identify the violation, and blame the violator. Identifying the faulty part of the system helps programmers narrow down the cause of the violation and, in a component-oriented setting, exposes culpable component producers.

Contracts enjoy widespread popularity. For example, contracts are currently the second most requested addition to Java.[1] In C code, assert statements are particularly popular, even though they do not have enough information to assign blame properly and thus are a degenerate form of contracts. In fact, 60% of the C and C++ entries to the 2005 ICFP programming contest [9] used assertions, even though the software was produced for only a single run.

[1] http://bugs.sun.com/bugdatabase/top25_rfes.do, as of Groundhog Day, 2007.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 111–128, 2008.

Despite the popularity of contracts, the state of the art in contract checking for data structures is poor. In order to use contracts on data structures, programmers are forced to choose between copied code (and thus doubled maintenance costs) and very poor performance (often infeasible, as we show). Our contract checker provides a new alternative. It is designed to strike a balance between performance and the amount of checking, motivated by the desire to avoid changing the asymptotic complexity of operations that have contracts. Our implementation is written in PLT Scheme [13], and is applicable to other strict languages with immutable data structures.

The next section uses binary search trees to make the programmer's existing poor choices plain. Section 3 explains the design of our contract checker and how it limits the amount of checking performed, in order to recoup tractable performance. Since our design is partially motivated by performance, we spend Section 4 explaining our implementation and Section 5 presenting some performance measurements that validate our design. For example, an experiment on binomial heaps show that eager checking may cause the program to be 2,000 to 20,000 times slower, while our lazy contract checker reduces that overhead to a factor between 8 and 10. Section 6 discusses an extension to our contract checker that relaxes the strict asymptotic complexity requirements; by giving the contract checker the freedom to preserve only the amortized complexity, we gain the ability to write more expressive contracts. Section 7 discusses related work and Section 8 concludes.

2 A Rock and a Hard Place

To see how existing techniques for data structure contracts fail programmers, consider a binary search tree library (shown in Figure 1) that is built on top of a binary tree library.

```
(module bt mzscheme
  (define-struct node (n left right))
   ...
  (provide (struct node (n left right)) marshal-bt unmarshal-bt))

(module bst mzscheme
  (require bt)
  ;; a Binary Search Tree (bst) is either null or
  ;; (make-node number[n] bst[left] bst[right])
  ;; where the numbers in left are less than (or equal to) n
  ;; and the numbers in right are greater (or equal to) n

  (provide find-bst) ;; : bst number → boolean
  (define (find-bst t n)
    (and (node? t)
         (or (= n (node-n t))
             (and (< n (node-n t))
                  (find-bst (node-left t) n))
             (and (> n (node-n t))
                  (find-bst (node-right t) n))))))
```

Fig. 1. Binary search trees, without contracts

The binary tree library is left mostly to the reader's imagination, but a skeleton is shown in the `bt` module.[2] It exports basic operations on binary trees (marshaling them to and from disk) and a node record for building and querying the nodes in a binary tree. In PLT Scheme, records are called structs. The **`define-struct`** introduces a new struct that consists of three fields. It also defines five functions: `make-node` used to build new nodes, `node?` used to recognize node structs, and `node-n`, `node-left`, and `node-right` used to extract the fields from a node struct. In general, a struct definition introduces a single maker, a single predicate, and one selector per field. The **`provide`** clause exports the struct and the marshaling functions.

The `bst` module requires the `bt` module and defines a binary search tree data structure in a comment, according to the discipline of *How to Design Programs* [7]. The comment specifies that binary search trees have the same shape and use the same node struct as binary trees, but also have the binary search tree invariant. The programmer carefully uses the same basic data structure so that the existing library for binary trees (marshaling and unmarshaling functions in this case) can also be used with binary search trees. Beyond the data definition, the `bst` module also provides `find-bst`, a function for finding numbers in binary search trees that takes advantage of the binary search tree invariant to avoid the recursive calls when it is safe to do so.

As the program grows from a little script to a part of a robust application, its author decides to improve the reliability of the program by writing a checkable contract on the data structure as shown in Figure 2. The `bst?` predicate uses the `bst-between?` helper function to test whether its input is a binary search tree. The function `bst-between?` enforces the binary search tree invariant using two accumulators, a lower and upper bound on the values in the tree. The accumulators are initially negative and positive infinity respectively, and as the traversal passes each interior node, the bounds tighten in the recursive calls.

Finally, the `bst?` predicate is used in the contracts for the provided functions.[3] The contract on `find-bst` is an $\rightarrow$ contract and is written using prefix notation. The last argument to $\rightarrow$ is a predicate on the result of `find-bst`, ensuring that it always produces booleans. The other two arguments are predicates on the inputs to `find-bst`, ensuring that the first argument is a binary search tree and that the second argument is a number. Similarly, the contract on `bst?` ensures that it is a predicate function.

Although it may not be obvious at first glance, the binary search tree portion of the revised library is now completely useless. In particular checking `find-bst`'s contract means that the `bst?` predicate is called on each argument supplied to `find-bst` in order to enforce the pre-condition (domain) contract. Since `bst?` traverses the entire tree, it ruins the optimization built into the `find-bst` function, changing the asymptotic complexity from logarithmic to linear, an exponential slowdown.

This state of affairs leaves the programmer in a bind; both the loss of performance and the loss of reliability are unacceptable. The conventional solution to this problem is to hide the raw struct operations behind an opaque module boundary and only export

[2] The `mzscheme` that appears after the module name is the language name of the module. MzScheme is PLT Scheme's implementation of the Scheme language.

[3] We use the PLT Scheme contract library's notation [22] throughout. Support for lazy data structure contracts was added to PLT Scheme's contract library in v350 (released June 2006).

```
(module bst mzscheme
  (require (lib "contract.ss"))

  ;; bst? : any → boolean
  (define (bst? t) (bst-between? t −∞ +∞))
  (define (bst-between? t low high)
    (or (null? t)
        (and (node? t)
             (number? (node-n t))
             (≤ low (node-n t) high)
             (bst-between? (node-left t) low (node-n t))
             (bst-between? (node-right t) (node-n t) high))))
  (define (find-bst t n) ...) ;; as in Figure 1

  (provide/contract
   [find-bst (→ bst? number? boolean?)]
   [bst? (→ any/c boolean?)]))
```

Fig. 2. Binary search trees, with contracts

operations that guarantee the binary search tree invariants (*e.g.*, self-balancing insert plus an empty binary search tree). Of course, this non-solution has the problem that a client of `bst` module cannot reuse the `bt` operations on `bst`s.

A programmer may attempt to work around this by providing new versions of each of the `bt` operations that simply unwrap a `bst` struct, apply the operation, and then rewrap it. This approach is not desirable for two reasons. Not only must the `bst` programmer anticipate all future extensions to the `bt` library, he must now also verify that none of the `bt` operations violate the binary search tree invariant, rather than letting the system itself ensure the binary search tree invariant holds.

Another solution is to provide injection and projection functions that convert binary search trees to and from binary trees and, along the way, verify the invariant. This solution amounts to changing the pre-condition on the `find` operation to a simple check, but requiring that programmers rewrite their programs to decide explicitly where to do the real checks. Worse, it is not always possible to avoid an asymptotic slowdown when binary search tree operations are interleaved with binary tree operations.

In general, code reuse is enabled by the ability to view a data structure with an invariant (like the binary search tree) as the same data structure but without the invariant. Or, put another way, code reuse is hindered by taking that ability away or allowing it only when accompanied by expensive invariant checks. Thus, the goal of this work is to provide a form of contract checking that allows programmers to view data structures with invariants as if they are just the underlying data structures, without any special action on the part of the programmer and without violating the invariant.

Throughout the remainder of this paper, we continue to use binary search trees as a motivating example. Nevertheless, our technique applies to many data structures that have invariants: heaps, self-balancing trees, sorted lists, etc. It is also useful whenever one wishes to use refinement types (but when a refinement type checker is not strong enough) such as even-length or non-empty lists, or viewing the result of Scheme's

`read` as having a particular shape. Another particularly fertile ground is a compiler's intermediate representation. Well-known intermediate representations like CPS and A-normal form [12] are easily expressed as contracts over the general expression type, and compiler authors who take advantage of them can determine which pass of a compiler has failed when bad output is produced.

3 Lazy Contract Checking

Our solution to the problem presented above is to introduce a new kind of contract for data structures to be used with the existing contract combinators in PLT Scheme. These contracts have the benefit of the contracts in Figure 2, namely they permit the programmer to use a single value with multiple, different contracts, but instead of eagerly checking the entire data structure when checking a contract, our contracts lazily check the portions of the data structure that the function inspects, as it inspects them.

Our contracts extend PLT Scheme's **`define-struct`** to **`define-contract-struct`**. It has the same syntactic shape as **`define-struct`**, but in addition to introducing a maker, predicate, and selectors, it also introduces a contract constructor. For example, the declaration

```
(define-contract-struct node (n left right))
```

introduces `node/dc`, the constructor for *node dependent contracts*. Its shape is

```
(node/dc [n contract-expr]
         [left (n) contract-expr]
         [right (n) contract-expr])
```

where each clause specifies the contract on the respective field. The `(n)` in the `left` and `right` contract specifications indicates that the contracts for the `left` and `right` fields depend on the value of the `n` field (the variables in the parenthesis are ordinary bound variables, but their names must match the names of other fields of the struct; that is, they may not be α-renamed). In general, the contract on any field may depend on any of the fields before it, but the dependencies must be specified explicitly by the programmer. Of course, `node/dc` is just one instance of a contract constructor; each **`define-contract-struct`** declaration introduces its own dependent contract constructor that expects as many fields as there are in the struct.

Using `node/dc`, the contract for a binary search tree is written as:

```
;; bounded-bst : number number -> contract
(define (bounded-bst lo hi)
  (or/c null?
        (node/dc [n (between/c lo hi)]
                 [left (n) (bounded-bst lo n)]
                 [right (n) (bounded-bst n hi)])))
(define bst (bounded-bst −∞ +∞))
```

The `or/c` contract combinator accepts any number of contracts (or simple predicates) and checks that at least one of them holds. The `between/c` contract combinator accepts

two numbers and returns a contract that matches numbers in those bounds. The contract on the left and right sub-trees of an interior node are built by recursively calling `bounded-bst` with different bounds on the values in the tree. The initial contract on a binary search tree is built by calling `bounded-bst` with negative and positive infinity.

The remainder of this section explains how dependent struct contracts behave, continuing to use the binary search trees example.

3.1 Checking During Traversal

The contract checker only checks struct contracts as the program itself inspects the data structure. To see how this plays out, consider this binary search tree and call to `find-bst`.

```
(define a-bst (make-node 5
                (make-node 3
                 (make-node 1 ...)
                 (make-node 6 null null))
                (make-node 7 ...)))
(find-bst a-bst 4)
```

The series of diagrams in Figure 3 shows the evolution of the contracts as `find-bst` traverses `a-bst` seaching for `4`. To represent the contract on the tree, we draw a box around the tree and annotate the box with the contract. So, when the tree is first passed to `find-bst`, it picks up the binary search tree contract and is labeled "$(-\infty,+\infty)$", meaning that the elements in the tree must be between $-\infty$ and $+\infty$, corresponding to the contract obtained by calling `(bounded-bst` $-\infty$ $+\infty$`)`. The first step `find-bst` takes is to examine the top node in the tree. At the point when `find-bst` first extracts a field of the top `node` struct, the contract checker steps in and verifies that the values of the fields of the node match the contract. Verifying that the number in the tree is in the appropriate range is a simple check, but to ensure that the subtrees match their contracts, the contract checker creates new boxes to avoid exploring more of the tree than the program does, as shown in Figure 3(b).

The labels on the new boxes indicate the new contracts, derived from the binary search tree invariant (as implemented by `bounded-bst`). The left sub-tree's elements

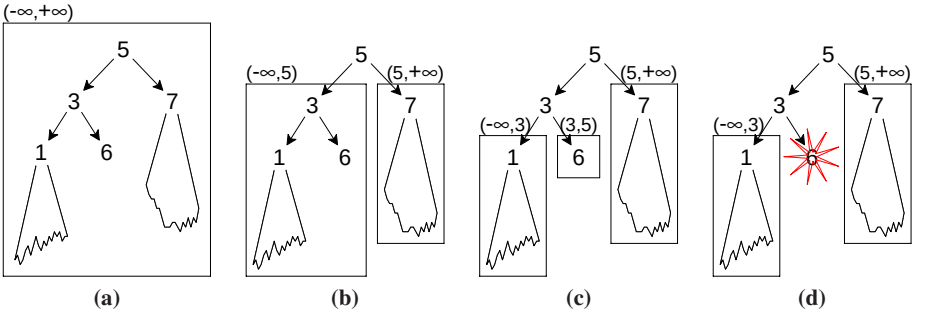


Fig. 3. Evolution of contracts during traversal of tree

must be smaller than 5 and the right sub-tree's elements must be larger than 5. Figure 3(c) shows the state of the tree after `find-bst` inspects the left child of the root. Again, the contract checker verifies that the node's value is appropriate and creates new boxes for the sub-trees. At this point in the program, no contract violation is signaled, because the program has not yet discovered the contract violation lurking one level down in the tree. Indeed, if the program never explores that part of the tree, a contract violation will never be signaled. But, because `find-bst` is searching for a 4, it does inspect that node, and a contract violation is signaled blaming the caller of `find-bst`.

3.2 Redundant Contracts

Although the boxes help eliminate much of the redundant work that eager contract checking would incur, it is still possible to do too much work. In particular, we must be careful to avoid accumulating multiple, redundant boxes on the same tree. To see how this happens, imagine that a tree is built up via an `insert : bst number` → `bst` operation that first calls `find-bst` to see if the value is in the tree and, if so, just returns the original tree. Consider the effect of these two calls during the evaluation of `(insert (insert a-bst 5) 5)`. Even though the two calls do not change the tree, a naive strategy for putting boxes on contracts accumulates surprisingly many new boxes.

Figure 4 shows what would happen for those two calls. Initially, the tree has no contracts, but as soon as it is passed to `insert`, the binary search tree contract is wrapped around it, as shown in Figure 4(a). The first thing `insert` does is pass the tree to `find-bst`, along with 5. Since 5 is in the root node of the tree, `find-bst` triggers the checking of only the first layer of the contracts, pushing contracts down to the left and right sub-children, and removing the outer layer of contracts. After that, the first call to `insert` returns and its post-condition adds another box around the entire tree and we are left with the tree shown in Figure 4(b).

As the second call to `insert` happens, the pre-condition adds another wrapper to the tree, leaving us with Figure 4(c). When `insert` calls `find-bst`, it inspects the top portion of the tree, pushing both of the contracts to its subtrees, and then the post-condition of `insert` adds yet another contract outside the tree, leaving us with Figure 4(d).

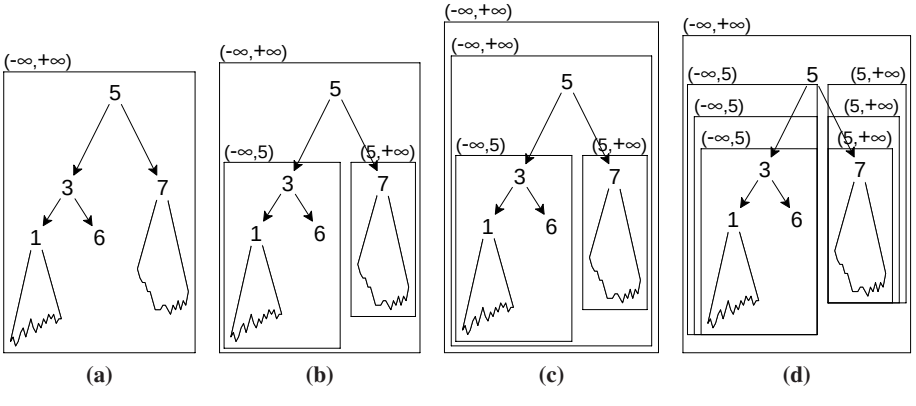


Fig. 4. Evolution of contracts during tree traversal without stronger check

To avoid this accumulation, we must be able to detect redundant contracts. In the case of a binary search tree, we can simply compare the bounds. If the box around a tree has the same (or tighter) bounds than what the new box would, then we can just leave the tree alone, relying on the existing contract to guarantee that the new contract holds.

To detect redundant contracts in general, our contract system supports a partial ordering on contracts that is used to compare two contracts to determine if one is stronger than or equal to the other. The ordering is tied to the particular contracts that our system supports. Each contract knows how to compare itself to certain other contracts in our system; if the contract does not recognize the other one, we avoid unsoundness by assuming that neither contract is stronger than the other.

As a design principle for our system, we decided that programmers who merely use contracts should not have the responsibility of specifying the stronger relationships. Instead, that responsibility should lie with the programmers that implement the contract combinators (such as `between/c`, $\rightarrow$, or the struct contracts). Accordingly, the stronger relationships are set in stone once a particular contract combinator has been defined. So far, this method has worked well enough for us, but we may also eventually investigate separating the stronger relation definition from the contract combinators and allowing programmers to extend it.

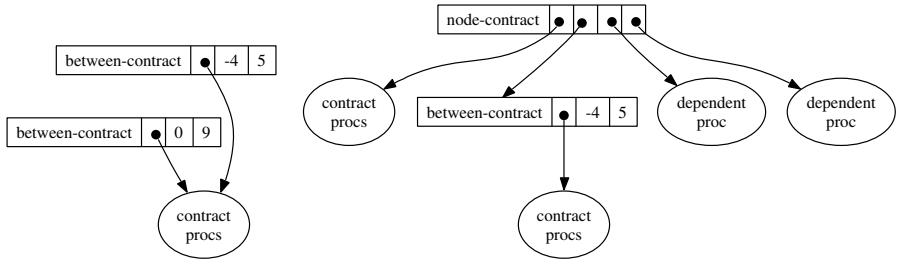
For `between/c` contracts, our system treats the one that accepts the same or a narrower range of numbers to be the stronger contract. One contract on a struct is stronger than another if the contracts on the fields of the first are stronger than the contracts on the fields of the second. Comparing function contracts uses the usual contra-variant ordering. To date, simple structural equality of contracts, combined with the bounds checking of `between/c` has been sufficient for all of the data structure invariants we have encountered (including all those in Okasaki's book [18] and in Cormen, Leiserson and Rivest [6]).

To exploit our new relation on contracts, we simply avoid adding a new contract if the contract already on the data structure is stronger than or equal to the new contract. Note that we do not need to consider blame here, unlike the case when the existent contract is not stronger; indeed, if two contracts surround a single data structure, the inner contract is always checked before the outer one, because the inner contract was placed on the object first. If the contract already on the data structure is stronger than the new contract, it does not matter who might be blamed if the new contract were to be violated; the existing contract guarantees it never fails.

Once we avoid adding redundant contracts, calling `insert` as above (or even arbitrarily many more times) would result in the wrappers shown in Figure 4(b). That is, each sub-tree would only have a single wrapper, no matter how many times `insert` is called.

4 Implementation and an Optimization

In our implementation, each contract is represented as a struct that has at least one field. That field contains a reference to a group of functions specific to that kind of contract that interpret the values in the other fields. The representation is inspired by the way



(a) Two between contracts **(b) A node contract**

Fig. 5. Contract layouts

objects are represented in class-based object-oriented languages: the record of functions is like the method table and is shared among every contract of a particular kind. As an example, Figure 5(a) shows a box-and-line diagram for the result of `(between/c −4 5)` and `(between/c 0 9)`. Each points to the same record of functions and has two numbers indicating the range it accepts.

A contract on a struct also has a shared record of contract procedures, but in addition it has one field per struct field. Each of those fields is either a contract that the contents of the field must satisfy directly, or it is a function that accepts the values in the other fields and returns such a contract. As an example, the contract

```
(node/dc [n (between/c −4 5)]
         [left (n) (bounded-bst −4 n)]
         [right (n) (bounded-bst n 5)])
```

is shown in Figure 5(b). The first field is the record of functions. Because the contract on the `n` field does not depend on other contracts, the second field of the contract record is the `between/c` contract. But, the `left` and `right` fields depend on the value of the `n` field, so they are functions that consume the `n` field's value and produce contracts.

Each contract's record of functions includes three functions. The first accepts the contract record and a value and enforces the contract. The second accepts two contracts and returns a boolean indicating whether the first is stronger than the second or not. The third function in the contract accepts a contract record and builds a name for the function to be used in error reporting.

To support lazy data structure contracts, we must not examine the struct's fields right away. Accordingly, the checking function for structs merely verifies that the struct's type matches, and then pairs the contract with the struct. Later, when a selector is applied to the struct, the contract is checked. Figure 6 contains a series of box and pointer diagrams that illustrate this process. The first diagram shows an example binary search tree, where the `null`s representing the empty tree are written `mt` to clarify the figure.

Figure 6(b) shows the tree paired with a between contract in a node-wrap struct. The node-wrap struct that holds the pair has a number of extra fields. The first field refers to the original object, but is also used as flag to indicate if the top row or the bottom row of fields are active. In the case shown, because that first field contains a reference to a struct, the top fields are active. Those fields contain a pointer to the contract, and two

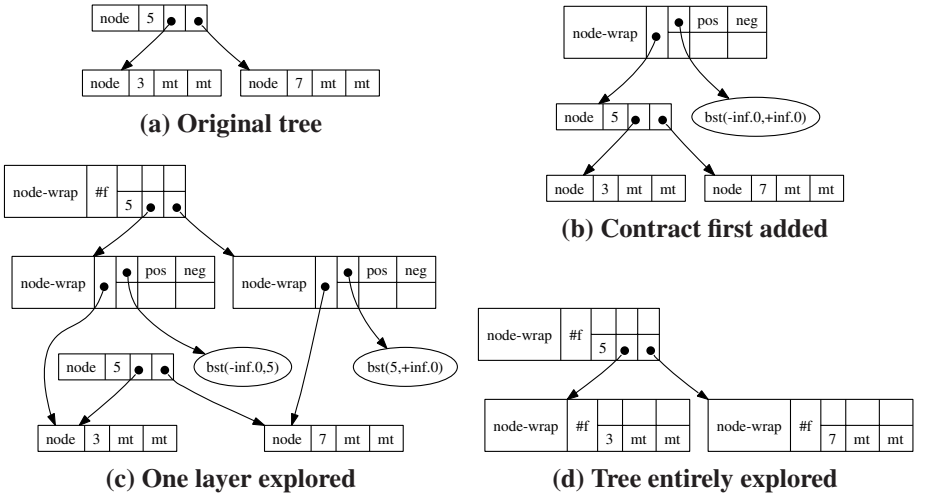


Fig. 6. Evolution of objects during contract checking

names that indicate who is to blame for contract violations. The label `pos` indicates who is to be blamed if this contract fails to hold, and `neg` is only used to support contract checking of functions that may appear inside this structure. It indicates the name of the party responsible for inputs to those functions [10,11]. Positive and negative infinity are written as `+inf.0` and `-inf.0`.

The other fields are used to implement the removal of the boxes described in Section 3. In particular, once the contract has been checked we know that it will continue to hold for all time, because the data structure is immutable. Accordingly, we place the contracted versions of the fields of the original struct into the bottom row of the node-wrap, to avoid recomputing them. When that happens, we also change the first field to `#f` in order to indicate that the bottom row is active.

Figure 6(c) shows the same tree, but after a selector has been applied to the struct with the contract, causing the contracts on the fields to be checked. The top node-wrap struct in this diagram is the same node-wrap struct in the top of diagram (b), but now the lower fields are active. The second field (in the bottom row) in that structure is `5`, the contracted version of the first field in the original struct. The final two fields are the contracted versions of the left and right sub-trees. The left sub-tree now has the contract `(bounded-bst` $-\infty$ `5)`, so it is a node-wrap struct whose first field is not `#f`. This node-wrap's top row is active, because its contract has not yet been checked. Similarly, the right sub-tree now has the unchecked `(bounded-bst 5` $+\infty$`)` contract. Finally, the fourth diagram shows the tree after all of the contracts have been checked. At this point, the tree is very similar to the original tree.

Since the top row and the bottom row are never simultaneously active for any given node-wrap struct, our implementation only has a single set of fields and uses the second field to indicate how to interpret the remaining fields.

Generally speaking, supporting the stronger relation for contracts is simply a matter of inspecting the structure of the contracts. For example, seeing if one `between/c`

contract is stronger than another amounts to comparing the numbers inside the contract record. The only exception to this is dependent struct contracts, when the fields actually are dependent. In that case, the dependent contracts are represented as functions that accept field values and return ordinary contracts. For example the `left` field of a node in the binary search tree contract is represented as the function

```
(λ (n) (bounded-bst lo n))
```

To compare such contracts, we exploit some information about the underlying representation of procedures in PLT Scheme. Specifically, we compare the contents of the closures corresponding to those functions (using simple pointer equality on the contents of the closure and the code pointer). In this case, the closure contains the free variables `bounded-bst` and `lo` and thus the closure will match any other closure that has the same value of `lo`, which suffices to avoid the redundancy seen in the example from Figure 4. Since this comparison may fail when standard compiler optimizations are performed, our implementation communicates with the compiler, telling it not to optimize these particular closures. So far, we have found this strategy for comparing contracts to be sufficiently powerful for the programs we have run. The next section discusses an experiment that demonstrates that our strategy has a significant, positive impact on the performance of our contract checker.

After some experimentation with our implementation, we discovered that a significant amount of time is spent in allocation, even with the stronger check in place. In particular, there is still significant extra allocation because the implementation allocates a record for each contract combinator. This approach becomes expensive when combined with dependent contract checking, because the allocation of the contracts happens during the traversal of the data structure. To compensate, we built a "flattening" optimization for lazy contracts that flattens nested contracts together into a single contract, in order to cut down on the allocation.

As an example, consider this contract:

```
(or/c null? (between/c 0 +∞))
```

It accepts either `null` or positive numbers. Without the optimization, the construction of this contract requires creating two records, one for the `or/c` contract and one for the `between/c` contract. With the optimization, we can simply create a single record that stores the bounds and simultaneously checks if the value is `null` or an appropriate number. Returning to Figure 5 (b), our optimization would only allocate a single record, replacing the two separate contract records with a single record for a node-between contract. Our optimization can also detect recursive contracts, so for the `bounded-bst` example, we can eliminate much of the allocation, requiring only a single allocation for each layer of the tree (to hold the new bounds).

5 Performance

This section presents the results of three experiments we performed on our implementation. Although these experiments are not conclusive, they do provide some validation of our contract checker. The first experiment validates the claims from Section 2 by showing that eagerly checking the contracts can be arbitrarily slower than lazily checking

them. The second experiment measures the cost of laziness, in the case that laziness is superfluous. The third demonstrates how our lazy contract checker behaves for more realistic applications and provides empirical evidence that it does indeed preserve the asymptotic complexity of the underlying operations.

We ran all of our experiments using PLT Scheme [13] v3.99.0.13 on a dual core 1.66 GHz Mac mini with 2 gigabytes of memory (although each test ran sequentially and only a test that disabled the stronger check allocated a significant amount of memory, discussed in Section 5.3).

5.1 The Cost of Eagerness

As we discussed earlier in this paper, the cost of eagerly checking data structure contracts can be arbitrarily bad. To verify this claim, we ran a simple test with our implementation. We built a toy program that constructs increasingly larger complete binary trees, numbers them via an inorder traversal (to satisfy the binary search tree invariant), and then measures the time it takes to search for each number.

Figure 7 shows the results. The x-axis ranges over the number of elements in the binary search trees, and the y-axis shows the slowdown as the *ratio* of the time required to call `find-bst` with the the eager contracts to the time to call `find-bst` with the lazy contracts. Each point on the graph represents a single run of each program. Even at the relatively modest size of a 10,000 element binary search tree, eager checking incurs an overhead that is more than 200 times greater than lazy checking. More worryingly, however, is the shape of the graph; as the size of the binary search tree increases, so does the factor of slowdown, meaning that eager checking is slowing down significantly more than lazy checking as the trees get bigger.

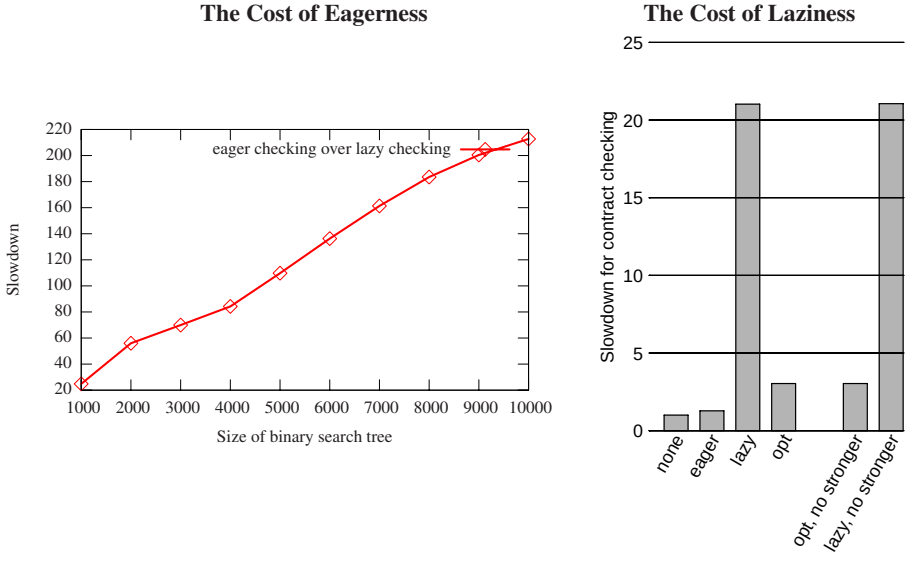


Fig. 7. Synthetic benchmark results

5.2 The Cost of Laziness

To measure the cost of laziness, we wrote a program that constructs a list of the numbers from 1 to 100,000. We did not use PLT Scheme's built-in `cons` function, because our contracts only support user-defined structs. Instead, we made a two field struct and used that for the pairs in the list. Once the list is built, the program applies different implementations of a contract that specifies that the list is sorted in ascending order, and then iterates over the list. Since the function always iterates over the entire list, delaying the contract does not improve the running time. Accordingly, this test helps us understand the cost of our implementation's bookkeeping. The right-hand side of Figure 7 shows those measurements. The height of each bar in the figure is the ratio of the performance of a particular contract to the performance of the code without any contracts.

The first four bars show the slowdown of the running time as compared to the version without contracts. The first bar (none) just gives a sense of scale; the slowdown for the version without contracts as compared to itself is 1. The second bar (eager) shows the slowdown for the eager contract that iterates down the entire list during the pre-condition checking, the third for the lazy contracts (lazy), and the fourth for lazy contracts with our flattening optimization (opt). Each bar corresponds to the average result of five runs. We see that the cost of the lazy contract bookkeeping is about a factor of 21 for this program, compared to a factor of 1.3 for the eager contract. Our optimization brings this cost down to a more reasonable factor of 3.0.

For a final experiment to measure the cost of laziness, we also set out to determine the cost of evaluating the stronger relation. For the program in this section, we know that no contract is ever going to be applied twice to the same object, so the stronger relation has no positive effect on the running time. We disabled the code that does that check and re-ran the tests. The results are shown as the final two bars in Figure 7. They show that the stronger check does not have a significant cost, when compared to the cost of the contract checking itself.

5.3 A Realistic Benchmark

For this experiment, we extracted traces of calls to a heap data structure from a colleague's vision algorithm [8]. We used four traces that are named after the images we used when extracting each trace: elephant, elephant-big, bird, and koala. The traces vary in size: elephant has roughly 22,000 inserts and 5,700 removals of minimum elements, whereas koala has more than 300,000 inserts and nearly 150,000 removals. We then coded up a binomial heap, as described in Okasaki's book [18] and ran the traces with three variations of the contracts on the heap operations: no contracts, optimized lazy contracts, and eager contracts. Accordingly, these results represent the times for only the data structure operations, not the original program that used the heap.

Figure 8(a) shows the slowdown for running the optimized lazy contract checking on heap operations; note that this chart's scale is not the same as that in Figure 7. As you can see, even though the traces vary in size, the overhead is relatively constant, encouraging us to believe that our contract checker only adds a constant overhead.

Figure 8(b) shows the slowdown for using the eagerly checked contracts on heap operations. Since these runs take a long time — running the koala trace once requires

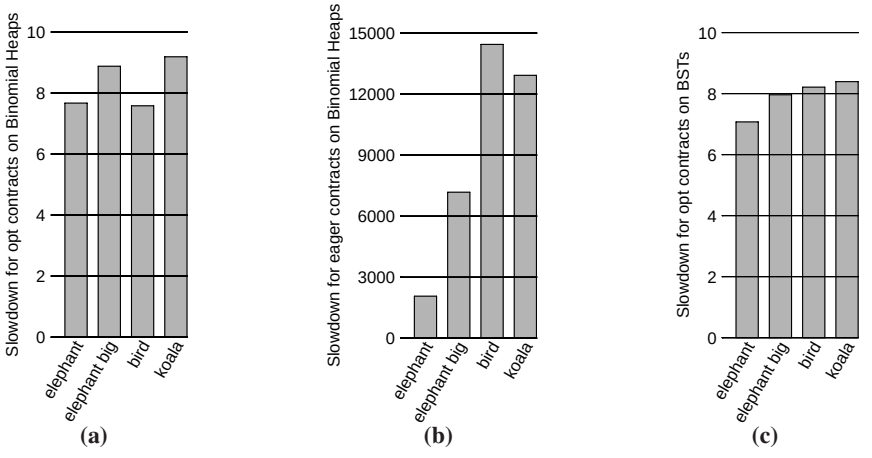


Fig. 8. Binomial heap and binary search tree experiments

about thirty CPU hours — we only ran them three times each. There are two important features of this chart. First, the scale is significantly different from that of the other two charts. The overheads are at least 2,000 and can be as bad as 14,400. Second, the overheads are not close to each other, demonstrating that the eager checking does not preserve the asymptotic complexity of the program.

We also synthesized traces for binary search trees from the heap traces. We replaced each heap insertion with a naive binary search tree insertion and replaced each extract minimum with a lookup of a random element in the tree. Figure 8(c) shows the slowdown when running the revised traces with the optimized contract checker and, as before, the overheads are relatively constant.

Finally, we performed one more experiment to test the contribution of the stronger check. We disabled the stronger check and then re-ran the optimized contract checker in the binary search tree experiment. Partway through the smallest trace, PLT Scheme had 1.5 gigabytes of resident storage (according to top) and then the machine proceeded to swap, making very little additional progress. This behavior indicates that a well-designed stronger relation is a crucial part of making the implementation practical.

6 Preserving Amortized Complexity

Implicit in the strategy of our lazy contract checker is a limitation of its expressiveness. In particular, the contract for the unexplored portion of the data structure must be expressible using only information in the explored portion of the data structure. This limitation is precisely what allows us to check the contract incrementally and to preserve the asymptotic complexity of the operations in the original program.

While this limitation still permits fairly expressive contracts, there are data structures with invariants that cannot be expressed. For example, one might wish to ensure that a binary tree is full, i.e., if the height of the tree is n, there are 2^n nodes in the tree. Intuitively, the contracts presented so far cannot express this contract because they require

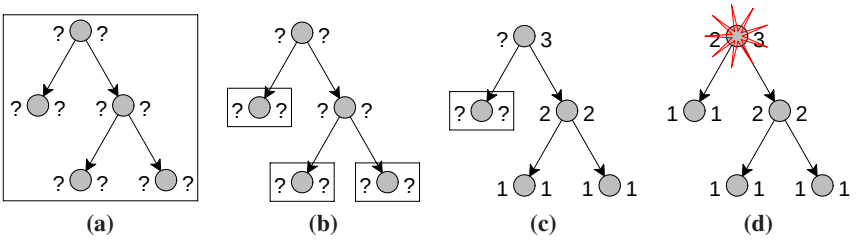


Fig. 9. Evolution of attributes during tree traversal for a full binary tree

knowledge of the particular height before reaching the leaf nodes. This restriction arises because the contracts thus far have only been based on values that are propagated "downwards", whereas fullness of a binary tree must be expressed with values that are propagated "upwards" as well. In the jargon of attribute grammars, the former are inherited attributes while the latter are synthesized attributes.

In order to check this contract, we must relax the strict constraint that contract checking will not affect the asymptotic complexity of the original program's operations. In particular, we allow the checker to preserve only the amortized asymptotic complexity of the program's operations while checking contracts that depend on the values of synthesized attributes. In order to check the full binary search tree contract, we can wait until the traversal reaches a leaf node and, at that point, propagate the height values to nodes on the path to the root.

To get a sense of how this kind of contract is checked, consider Figure 9. This contract has two synthesized attributes: a left height and a right height, denoting the height of a node's left and right children. The invariant is that both heights must be equal once they are known. Like ordinary struct contracts, contracts with attributes are checked lazily. As in Figure 3, we box the uninspected portions of the tree. Initially, each node is decorated with two question marks, indicating that the values of the left and right heights are both unknown. Figure 9(b) shows the state of the tree after inspecting the two interior nodes. At this point, none of the attribute's values are known, because none of the leaf nodes have been discovered. In Figure 9(c), the program inspects the children in the rightmost subtree. Since they have no children, their left and right height attributes are both `1`. At this point, because some attribute values have become known, propagation is triggered, resulting in the `right-height` attribute of the root becoming `3`. Finally, in Figure 9(d), the program inspects the left-most child, triggering propagation of the left height back to the root, where a contract violation is discovered, because the left height and right height of the root are not the same.

Our contract system takes care of the propagation and verification of these attributes as well as determining whether they are known or unknown. The programmer, on the other hand, must provide the logic of how those attributes are computed and what to do with them once they are known. In Figure 9, for example, our system takes care of the boxing and the propagation of the tree heights back up the tree, but it is the programmer who determines that those attributes are to be propagated upon discovering leaf nodes and that both left and right heights must be equal once they are both known. This

separation allows enough expressiveness to implement more complex contracts than our motivating example while still preserving the amortized asymptotic complexity.

Since each wrapper has a fixed c number of attributes, the propagation can occur at most c times, and each node in the tree will be inspected at most c times. Thus, the complexity of the program can only change by the constant factor, c. Because attribute evaluation may propagate an unbounded distance when just a single field is selected, however, only the amortized complexity of the original operations in the program is preserved.

7 Related Work

The idea of software contracts dates back to the 1970s [19]. In the 1980s, Meyer developed an entire philosophy of software design based on contracts, embodied in his object-oriented programming language Eiffel [16]. Nowadays, contracts are available in one form or another for many programming languages (e.g., C [23], C++ [21], Haskell [14], Java [15], Perl [5], Python [20], Scheme [22], and Smalltalk [1]).

Although the authors did not make the connection until much of this work had been done, this work is a direct intellectual descendent of Okasaki's dissertation [17], where Okasaki demonstrates that a controlled amount of laziness, in an otherwise strict language, makes achieving desired asymptotic bounds tractable. We cannot, however, use Okasaki's `$` operator directly, because we need fine-grained control over the laziness to exploit the stronger relation.

From a contracts perspective, our work is anticipated by Chitil, McNeill, and Runciman's and Chitil and Huch's Lazy Assertions [2,3,4]. They observe that eagerly checking assertions in a lazy setting can introduce non-termination where none should rightly be. In particular, a strict assertion on an infinite list should not explore the entire list unless the program itself explores the entire list. They attempt to preserve laziness in a lazy world, whereas our work attempts to add laziness to a strict world. Despite starting from very different foundations, both arrive at the conclusion that laziness for checking contracts on data structures is necessary. From a technical point of view, we believe that the stronger relation should carry back to their setting and should help them with memory use, and that the ideas in Section 6 should also apply to their system.

Hinze, Jeuring, and Löh's contract checker [14] is also a contract checker for Haskell (that correctly handles blame), but their checker explores parts of the data structure that the program does not. For example, the `is(sort)` example contract in Section 6 of their paper explores the entire list; a similar contract in our system would not.

Beyond that, there is little other work on checking data structure contracts, except when using naive strategies. Eiffel, the language most focused on contract checking, provides no native support for lazy contract checking. Tremblay and Cheston [24] wrote an algorithms and data structures textbook using Eiffel, but the contracts in their text either only partially check the data structure invariants or check them as the data structure is constructed.

8 What Have We Gained?

In some sense, this work puts data structure contract checking on an even footing with function-based contract checking. Specifically, when checking a contract on a function,

violations can go undetected if the function is never called with an input that would trigger an error. Similarly, consider this (supposed) binary search tree:

```
(make-node 5
           (make-node 7 null null)
           (make-node 207 null null))
```

If `find-bst` is called with that tree and, say, `6`, the contract checker will not discover the violation. Even worse, if it is called with `7`, `find-bst` will indicate that `7` is not in the binary search tree, and the contract checker will still fail to detect the violation. Of course, similar behavior can happen with functions (in fact, this binary search tree could be encoded as a function to achieve precisely the same behavior) and yet function contracts enjoy wide-spread use.

We believe that our data structure contracts have the potential to enjoy similar wide-spread use, for two reasons. First, it is rare for a data structure to be built that will not eventually be completely explored in a long-running application. Even though the two calls to `find-bst` above do not detect the violation, it seems likely that some later call to `find-bst` will ask for a number smaller than `5`, resulting in a contract violation.

Second, our checker makes checking data structure contracts feasible. As discussed in Section 5.3, using either the naive strategy of eagerly checking the contracts, or even avoiding the stronger check makes checking the contracts infeasible, for at least one realistic program. Intuitively, we expect the naive strategy to fail in general, simply because the change to the asymptotic complexity incurred by the naive checker is a tremendous expense.

Fundamentally, the question we ask is how much contract checking can we expect a program to be able to afford? Our contract checker represents one answer to this question that does not take into account any a priori knowledge about the program's behavior; it provides a maximal amount of contract checking that we can reasonably expect the program to be able to afford, namely a constant factor.

Acknowledgements. Thanks to Ryan Culpepper and Matthew Flatt for PLT Scheme infrastructure support and to Matthew for comments on drafts of the paper. Thanks to Pedro Felzenszwalb for supplying us with the heap traces we use in our experiments. Also, thanks to Matthias Felleisen, Simon Peyton Jones, and Jacob Matthews for several enlightening discussions and comments on this paper. This work is supported in part by the NSF.

References

1. Carrillo-Castellon, M., Garcia-Molina, J., Pimentel, E., Repiso, I.: Design by contract in Smalltalk. Journal of Object-Oriented Programming 7(9), 23–28 (1996)
2. Chitil, O., Huch, F.: A pattern logic for prompt lazy assertions. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 126–144. Springer, Heidelberg (2007)
3. Chitil, O., Huch, F.: Monadic prompt lazy assertions in Haskell. In: Asian Symposium on Programming Languages and Systems (2007)
4. Chitil, O., McNeill, D., Runciman, C.: Lazy assertions. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 1–19. Springer, Heidelberg (2004)

5. Conway, D., Goebel, C.G.: Class: Contract – design-by-contract OO in Perl, http://search.cpan.org/~ggoebel/Class-Contract-1.14/
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
7. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs. MIT Press, Cambridge (2001), http://www.htdp.org/
8. Felzenszwalb, P., McAllester, D.: A min-cover approach for finding salient curves. In: IEEE Workshop on Perceptual Organization in Computer Vision (2006), http://people.cs.uchicago.edu/~pff/papers/
9. Findler, Barzilay, Blume, Codik, Felleisen, Flatt, Huang, Matthews, McCarthy, Scott, Press, Rainey, Reppy, Riehl, Spiro, Tucker, Wick: In: The eighth annual ICFP programming contest, http://icfpc.plt-scheme.org/
10. Findler, R.B., Blume, M.: Contracts as pairs of projections. In: International Symposium on Functional and Logic Programming, pp. 226–241 (2006)
11. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of ACM SIGPLAN International Conference on Functional Programming, pp. 48–59 (2002)
12. Flanagan, C., Sabry, A., Duba, B., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (1993)
13. Flatt, M.: PLT MzScheme: Language manual. Technical Report PLT- TR05-1-v300, PLT Scheme Inc. (2005), http://www.plt-scheme.org/techreports/
14. Hinze, R., Jeuring, J., Löh, A.: Typed contracts for functional programming. In: International Symposium on Functional and Logic Programming (2006)
15. Karaorman, M., Hölzle, U., Bruno, J.: jContractor: A reflective Java library to support design by contract. In: Cointe, P. (ed.) Reflection 1999. LNCS, vol. 1616. Springer, Heidelberg (1999)
16. Meyer, B.: Eiffel: The Language. Prentice Hall, Englewood Cliffs (1992)
17. Okasaki, C.: Purely Functional Data Structures. PhD thesis, Carnegie Mellon University, Technical Report CMU-CS-96-177 (September 1996)
18. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1999)
19. Parnas, D.L.: A technique for software module specification with examples. Communications of the ACM 15(5), 330–336 (1972)
20. Plösch, R.: Design by contract for Python. In: IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference (1997), http://citeseer.nj.nec.com/257710.html
21. Plösch, R., Pichler, J.: Contracts: From analysis to C++ implementation. In: Technology of Object-Oriented Languages and Systems, pp. 248–257 (1999)
22. PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR2007-4-v372, PLT Scheme Inc. (2007), http://www.plt-scheme.org/techreports/
23. Rosenblum, D.S.: A practical approach to programming with assertions. IEEE Transactions on Software Engineering 21(1), 19–31 (1995)
24. Tremblay, J.-P., Chesterton, G.A.: Data Structures and Software Development in an Object-Oriented Domain: Eiffel Edition. Prentice Hall, Englewood Cliffs (2001)

The Reduceron: Widening the von Neumann Bottleneck for Graph Reduction Using an FPGA

Matthew Naylor and Colin Runciman

University of York, UK
{mfn,colin}@cs.york.ac.uk

Abstract. For the memory intensive task of graph reduction, modern PCs are limited not by processor speed, but by the rate that data can travel between processor and memory. This limitation is known as the *von Neumann bottleneck*. We explore the effect of *widening* this bottleneck using a special-purpose graph reduction machine with wide, parallel memories. Our prototype machine – the Reduceron – is implemented using an *FPGA*, and is based on a *simple* template-instantiation evaluator. Running at only 91.5MHz on an FPGA, the Reduceron is faster than mature bytecode implementations of Haskell running on a 2.8GHz PC.

1 Introduction

The processing power of PCs has risen astonishingly over the past few decades, and this trend looks set to continue with the introduction of multi-core CPUs. However, increased processing power does not necessarily mean faster programs! Many programs, particularly *memory intensive* ones, are limited by the rate that data can travel between the CPU and the memory, not by the rate that the CPU can process data.

A prime example of a memory intensive application is *graph reduction* [14], the operational basis of standard lazy functional language implementations. The core operation of graph reduction is *function unfolding*, whereby a function application $f\ a_1 \cdots a_n$ is reduced to a fresh copy of f's body with its free variables replaced by the arguments $a_1 \cdots a_n$. On a PC, unfolding a single function in this way requires the sequential execution of *many* machine instructions. This sequentialisation is merely a consequence of the PC's von Neumann architecture, *not* of any data dependencies in the reduction process.

In an attempt to improve upon the PC's overly-sequential approach to function unfolding, we develop a special-purpose graph reduction machine – the Reduceron – using an FPGA. Modern FPGAs contain hundreds of independent memory units called *block RAMs*, each of which can be accessed in parallel. The Reduceron *cascades* these block RAMs to form separate dual-port, quad-word memories for stack, heap and combinator storage, meaning that up to eight words can be transferred between two memories in a single clock cycle. Together

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 129–146, 2008.

with vectorised processing logic, the wide, parallel memories allow the Reduceron to rapidly execute the block read-modify-write memory operations that lie at the heart of function unfolding.

The Reduceron is only a *prototype* machine, based on a *simple* template-instantiation evaluator, and the Reduceron compiler performs *no optimisation*. Yet the results are promising. The *wide* implementation of the Reduceron runs six times faster than a single-memory, one-word-at-a-time version. And running at 91.5MHz on a Xilinx Virtex-II FPGA, the Reduceron is faster than mature bytecode implementations of Haskell running on a Pentium-4 2.8GHz PC.

This paper is structured as follows. Section 2 defines the bytecode that the Reduceron executes, and describes how Haskell programs are compiled down to it. Section 3 presents a small-step operational semantics of the Reduceron, pinpointing the parts of the evaluator that can be executed in parallel. Section 4 describes the FPGA implementation of the Reduceron. Section 5 presents performance measurements, comparisons and possible enhancements. Section 6 discusses related work, and section 7 concludes.

The source code for the Reduceron implementation is publicly available from `http://www.cs.york.ac.uk/fp/darcs/reduceron2`.

2 Compilation from Haskell to Reduceron Bytecode

There are two main goals of our compilation scheme. First, it should allow the Reduceron to be *simple*, so that the implementation can be constructed in good time. To this aim, we adopt the idea of Jansen to encode data constructors as functions and case expressions as function applications [9]. The result is that all data constructors and case expressions are eliminated, meaning fewer language constructs for the machine to deal with. One might expect to pay a price for this simplicity, yet Jansen's interpreter is rather fast in practice. It is believed that one reason for this good performance is that having fewer language constructs permits a simpler interpreter with less interpretive overhead.

The second goal of our compiler is to expose the *parallelism* present in sequential graph reduction. An earlier version of the Reduceron was based on Turner's combinators, so it performed only a small amount of work in each clock cycle. Our aim is to do lots of work in each clock cycle, so the coarser-grained *supercombinator* [8] approach to graph reduction is taken here.

Like Jansen's interpreter, the graph reduction technique used by the Reduceron is similar to what Peyton Jones calls *template instantiation* [14]. Peyton Jones introduces template instantiation as a "simple" first step towards a more sophisticated machine – the G-machine. In this light, the Reduceron might be seen as being too far from a "real" functional language implementation to produce meaningful results. But Jansen's positive results give good reason to be open-minded about this.

The remainder of this section describes in more detail the stages of compilation to get from Haskell programs to Reduceron bytecode. As a running example we use the following function for computing the factorial of a given integer:

```
fact :: Int -> Int
fact n = if n == 1 then 1 else n * fact (n-1)
```

We end this section by defining the Reduceron bytecode and showing how `fact` looks at the bytecode level.

2.1 Desugaring and Compilation to Supercombinators

The first stage of compilation is to translate the input program to *Yhc Core* [6] using the York Haskell Compiler [17]. The result is an equivalent but simplified program in which expressions contain only function names, function applications, variables, data constructions, case expressions, let expressions, and literals. All function definitions are *supercombinator* definitions. In particular, they do not contain any lambda abstractions. In our example, `fact` is already a supercombinator, but in Yhc Core its definition becomes:

```
fact n = case (==) n 1 of
           True -> 1
           False -> (*) n (fact ((-) n 1))
```

Here, infix applications have been made prefix, and the `if` expression has been desugared to a `case`.

2.2 Eliminating Data Constructors and Cases

The second stage eliminates all data constructions and case expressions from the program. First, each data type d of the form

$$\texttt{data}\ d\ \texttt{=}\ c_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n$$

is replaced by a set of function definitions, one for each data constructor c_i, of the form

$$c_i\ v_1\ \cdots\ v_{\#c_i}\ w_1\ \cdots\ w_n\ \texttt{=}\ w_i\ v_1\ \cdots\ v_{\#c_i}$$

where $\#c$ denotes the number of arguments taken by the constructor c. In words, each original data constructor c_i is encoded as a function that takes as arguments the $\#c_i$ arguments of c_i *and* n continuations stating how to proceed depending on the constructor's value.

Next, all default alternatives in case expressions are removed. Case expressions in Yhc Core already have the property that the pattern in each alternative is at most one constructor deep. So removing case defaults is simply a matter of enumerating all unmentioned constructors. Now each case expression has the form

$$\texttt{case}\ e\ \texttt{of}\ \{c_1\ v_1\ \cdots\ v_{\#c_1}\ \texttt{->}\ e_1\ ;\ \cdots\ ;\ c_n\ v_1\ \cdots\ v_{\#c_n}\ \texttt{->}\ e_n\}$$

and can be straightforwardly translated to a function application

$$e\ (\lambda v_1\ \cdots\ v_{\#c_1}\ \texttt{->}\ e_1)\ \cdots\ (\lambda v_1\ \cdots\ v_{\#c_n}\ \texttt{->}\ e_n)$$

Since this transformation reintroduces lambda abstractions, the lambda lifter is reapplied to make all function definitions supercombinators once again. After this stage of compilation, our factorial example looks as follows:

```
fact n = (==) n 1 1 ((*) n (fact ((-) n 1)))
```

2.3 Dealing with Strict Primitives

Further to user-defined algebraic data types, the Reduceron also supports, as *primitives*, machine integers and associated arithmetic operators. Under lazy evaluation, primitive functions, such as integer multiplication, need special treatment because their arguments must be fully evaluated before *before* they can be applied. Peyton Jones and Jansen both solve this problem by making their evaluators recursively evaluate each argument to a primitive. This is an elegant approach when the evaluator is written in a programming language like Miranda or C, where the presence of an implicit call stack may be assumed. But FPGAs have no such implicit call stack, so an alternative solution must be found.

Our solution is to treat primitive values in the same way as nullary constructors of an algebraic data type: they become functions that take a continuation as an argument. The idea is that the continuation states what to do once the integer has been evaluated, and it takes the fully evaluated integer as its argument. Transforming the program to obtain this behaviour is straightforward. Each two-argument primitive function application is rewritten by the rule

$$p\ n\ m \ \rightarrow \ m\ (n\ p)$$

The factorial function is now:

```
fact n = 1 (n (==)) 1 (fact (1 (n (-))) (n (*)))
```

2.4 Reduceron Bytecode

In the final stage of compilation, programs are turned into Reduceron bytecode. The bytecode for a program is defined to be a sequence of *nodes*, and the syntax of a node is defined in Figure 1. In the syntax definition, the meta-variables i and p range over integers and primitive function names respectively.

An n-ary application node (`Ap` i) in Reduceron bytecode is a pointer i to a sequence of n consecutive nodes in memory whose final node is wrapped in an `End` marker. To permit sharing in over-saturated applications, the nodes in an application sequence are stored in *reverse* order, e.g. $f\ x\ y$ would be stored as $y\ x$ (`End` f), and if $f\ x$ evaluates to z then the application can simply be updated to y (`End` z) without relocating it in memory. To illustrate, Figure 2 shows the bytecode of the `fact` function, as it would appear relative to some address a in program memory. Each application node in the bytecode is an *offset* address, relative to a, the address of the first node of the function's bytecode. This first node is always a `Start` node, and defines the arity and size (number of words) of the function's body. The bytecode for a whole program is simply the concatenation of the bytecodes for each individual function. Each `Fun` node is then adjusted to point to the final location of the function in the program.

node	::=	`Start` i i	(first node of a function body: arity and size of function)
	|	`Int` i	(primitive integer)
	|	`Ap` i	(application node: a pointer to a sequence of nodes)
	|	`End` *node*	(the final node in a node sequence)
	|	`Prim` p	(primitive function name)
	|	`Fun` i	(pointer to a function body)
	|	`Var` i	(variable representing a function argument)

Fig. 1. The syntax of nodes in Reduceron Bytecode

a	$+1$	$+2$	$+3$
`Start 1 15`	`Ap 7`	`Int 1`	`Ap 5`
$+4$	$+5$	$+6$	$+7$
`End (Int 1)`	`Prim (==)`	`End (Var 0)`	`Ap 12`
$+8$	$+9$	$+10$	$+11$
`Ap 10`	`End (Fun` a`)`	`Ap 14`	`End (Int 1)`
$+12$	$+13$	$+14$	$+15$
`Prim (*)`	`End (Var 0)`	`Prim (-)`	`End (Var 0)`

Fig. 2. The bytecode for `fact`, as it would appear relative to address a in memory

3 An Operational Semantics for the Reduceron

In this section, a semantics for the Reduceron is defined. The are two reasons for presenting a semantics: first to define precisely how the Reduceron works, and second to highlight the parts of the reduction process that can be assisted by *special-purpose hardware*. The semantics is given as a binary small-step state transition relation, $\Rightarrow$, between triples of the form $\langle h,\ s,\ a\rangle$, where h is the heap, s is the node stack, and a is the address stack.

In defining the semantics, we model the heap and stacks as *lists*, and assume the availability of several common functions on lists. In addition, we write $\#xs$ to denote the *length* of the list xs and $xs[i \mapsto x]$ to denote xs with its i^{th} element replaced by x.

Initially, the heap contains the bytecode of the program, the node stack contains the node `Fun 0`, where 0 is the address of the function `main :: Int`, and the address stack contains the address 0. The final result of a program p is defined to be r where

$$\langle p,\ [\texttt{Fun}\ 0],\ [0]\rangle\ \Rightarrow^{\star}\ \langle \_,\ [\texttt{Int}\ r],\ \_\rangle$$

Notice that the `main` function is a pure (non-monadic) function with no arguments. Currently, Reduceron programs take no input. Furthermore, their output is a single integer – `main` is of type `Int`.

3.1 A Primitive Evaluator

We assume a function $\mathcal{P}$ that takes a primitive function name p and two integers, i and j, and returns a *node* representing the value of $p\ i\ j$. For example,

$$\mathcal{P}\ \texttt{(+)}\ 5\ 10\ =\ \texttt{Int}\ 15$$

and

$$\mathcal{P}\ \texttt{(==)}\ 1\ 1\ =\ \texttt{Fun}\ \mathit{true}$$

where *true* is the address of the function `True` in the bytecode of the program.

3.2 Semantic Definition

The small-step transition relation $\Rightarrow$ is defined in Figure 3 and the helper functions *inst* and *unwind* are defined in Figure 4. There is one transition rule for each possible type of node that can appear on top of the stack, as described by the following paragraphs.

$$\langle h,\ \texttt{Int}\ i : x : s,\ a\rangle \Rightarrow \langle h,\ x : \texttt{Int}\ i : s,\ a\rangle$$

$$\langle h,\ \texttt{Prim}\ p : \texttt{Int}\ x : \texttt{Int}\ y : s,\ \_ : \_ : r : a\rangle \Rightarrow \langle h[r \mapsto \texttt{End}\ z],\ z : s,\ r : a\rangle$$
$$\textbf{where}$$
$$z = \mathcal{P}\ p\ x\ y$$

$$\langle h,\ \texttt{Ap}\ i : s,\ \_ : a\rangle \Rightarrow \mathit{unwind}\ i\ \langle h,\ s,\ a\rangle$$

$$\langle h,\ \texttt{Fun}\ i : s,\ a\rangle \Rightarrow \mathit{unwind}\ \#h\ \langle h',\ s',\ a'\rangle$$
$$\textbf{where}$$
$$\begin{aligned}
\texttt{Start}\ \mathit{arity}\ \mathit{size} &= h\ !!\ i\\
\mathit{body} &= \mathit{take}\ \mathit{size}\ (\mathit{drop}\ (i+1)\ h)\\
s' &= \mathit{drop}\ \mathit{arity}\ s\\
r : a' &= \mathit{drop}\ \mathit{arity}\ a\\
h' &= h[r \mapsto \texttt{End}\ (\texttt{Ap}\ \#h)]\\
&\qquad +\!\!+\ \mathit{map}\ (\mathit{inst}\ s\ \#h)\ \mathit{body}
\end{aligned}$$

Fig. 3. Transition rules for the Reduceron

Primitives. Recall from section 2.3 that primitive applications of the form $p\ a\ b$, where a and b are unevaluated integers, are transformed to $b\ (a\ p)$. Clearly, to evaluate such an application, b must be evaluated first. This results in the *value* of b, of the form `Int` i, appearing on top of the stack. To deal with such a situation, the evaluator simply *swaps* the top two stack elements, resulting in $(a\ p)\ b$ on the stack. Further evaluation yields $a\ p\ b$ on top of the stack and then, after another swap, $p\ a\ b$, where a and b are now fully evaluated, and evaluation of the primitive application is straightforward.

Once the result of the primitive application has been computed, it must be written onto the heap, overwriting the the contents of the original application node $b\ (a\ p)$, so that other references to it do not repeat the computation. This is possible because, as will be explained shortly, a pointer to the original application is sitting on the *address stack*.

Applications. When an application node of the form `Ap` i appears on top of the stack, it is replaced by the `End`-terminated sequence of nodes starting at address i on the heap. Furthermore, the addresses of the nodes in the sequence are pushed on the address stack, to permit updating the sequence after reduction. Following Peyton Jones's terminology, we collectively call these two tasks *unwinding*.

In an implementation of the Reduceron on a standard PC architecture, each node in an application sequence is read, one at a time, from the heap and written, one at a time, to the stack. Furthermore, each node address is computed and written, again one at a time, onto the address stack.

The definition of the *unwind* function in Figure 4 highlights the first main opportunities for hardware-assisted graph reduction. First, the uses of *getAp* and $+\!\!+$ illustrate that the nodes being copied are *contiguous*, so the copying can be achieved by block transfers in a machine with a wider data bus. Second, the

$$
\begin{array}{rcl}
inst\ s\ b\ (\texttt{Var}\ i) & = & s\ !!\ i \\
inst\ s\ b\ (\texttt{Ap}\ i) & = & \texttt{Ap}\ (b+i-1) \\
inst\ s\ b\ (\texttt{End}\ n) & = & \texttt{End}\ (inst\ s\ b\ n) \\
inst\ s\ b\ n & = & n \\
\\
unwind\ i\ \langle h,\ s,\ a\rangle & = & \langle h,\ reverse\ ap +\!\!+ s,\ reverse\ as +\!\!+ a\rangle \\
\textbf{where} & & \\
ap & = & getAp\ (drop\ i\ h) \\
as & = & map\ (i+)\ [0 \ldots \#ap - 1] \\
\\
getAp\ (\texttt{End}\ n : ns) & = & [n] \\
getAp\ (n : ns) & = & n : getAp\ ns
\end{array}
$$

Fig. 4. Definitions of *inst* and *unwind*

use of *map* to compute the node addresses indicates that they can be computed in parallel. And third, there is no dependency between writing to the node and address stacks, so the two can be done at the same time in a machine with parallel memories.

Functions. When a node of the form `Fun i` is at the top of the stack, the bytecode starting at address $i + 1$ on the heap is

1. *copied* onto the end of the heap (say at address *hp*),
2. with each variable `Var` j *substituted* with the j^{th} argument on the stack,
3. and with each application node, `Ap` k, *relocated* to an absolute address, `Ap` $(hp + k - 1)$, on the heap.

Subsequently, n nodes are popped off the node and address stacks, where n is the arity of the function that has just been instantiated. The address r which is n places from the top of the address stack represents the *root* of the redex. The value at r is overwritten with `End` (`Ap` *hp*), so that the reduction is never repeated. Finally, the node sequence beginning at the address *hp* is unwound onto the stack. We refer to this whole collection of operations as *function unfolding*.

Just as for unwinding, function unfolding on a standard PC architecture requires execution of many sequential instructions to carry out all the necessary memory manipulations. And again the semantics shows great scope for parallelism. In particular, the use of ++ to copy a potentially large contiguous block of nodes onto the end of heap, and the use of *map* to instantiate each node independently, opens up the possibility for parallelisation on a machine with wide memory and vectorised processing logic. Since instantiation of a node requires access to the stack, a parallel evaluator would need to be able to read the stack and heap at the same time. Further, because the nodes are being copied from one portion of memory to another, sequentialisation can be reduced by separating program memory and application node memory, permitting parallel access.

Notice in the semantics that the `Fun` rule calls *unwind*. Immediately after a combinator body is instantiated on the heap, the *spine* of that body is unwound from the heap onto the stack. It is much more efficient to instantiate the spine of the combinator on the heap and the stack *in parallel*. This idea is related to the *spineless* G-machine [2], which, by keeping track of which nodes are not *shared*, can often completely bypass construction of the combinator spine on the heap. So our idea gives the speed benefit of the spineless G-machine without introducing any complexity, but not the space benefit.

4 Implementation on FPGA

The semantics presented in the previous section suggests that an efficient implementation of the Reduceron can be obtained if *wide, parallel memories* and *vectorised processing logic* are available. A suitable architecture on which to explore this possibility is the FPGA. FPGA devices are ideal for constructing custom processing logic and typically contain large arrays of independent block

RAMs. This section describes our implementation of Reduceron on the FPGA device available to us, a Xilinx Virtex-II.

To measure of the effect of our proposed optimisations, we implement two versions of the Reduceron on the Virtex-II: *Baseline* and *Wide*. The Wide version exploits wide, parallel memories and the Baseline version does not.

4.1 Block RAMs

The Virtex-II contains 56 independent 1024 by 18-bit dual-port block RAMs. Being "dual-port" means that a RAM has two address busses, two data busses and two write enable signals. Thus two different locations in RAM can be accessed in a single clock cycle. Furthermore, each port has separate busses for data input and data output. Thus a value may be written to and read from a single location on a single RAM port at the same time. The *Wide* Reduceron exploits both the dual-port and separate data bus features of block RAMs, and the *Baseline* version does not. Both versions of the Reduceron encode bytecode nodes as 18 bit words, so each RAM location has capacity for a single node.

4.2 Constructing Large Memories from Small Ones

The Baseline Reduceron *cascades* 48 block RAMs to form a single 48k word memory; 32k is used as heap and stack memory and 16k is used solely by the garbage collector (described in section 4.5). Block RAMs are cascaded in the standard way using a multiplexor to combine the outputs of several memories into a single output. When cascading a large number of block RAMs the multiplexor becomes rather large and its delay becomes significant. To overcome this inefficiency, a register is placed on the output of the multiplexor. This means that two clock cycles are needed between writing an address to the address-bus and reading the resulting value off the data-bus. This overhead is alleviated by *pipelining*, whereby a new memory access is scheduled while waiting for the previous one to complete. But keeping the pipeline primed at all times is difficult, so some overhead is inevitable. Such overhead is present in both versions of Reduceron, as we always buffer RAM outputs in a register.

4.3 Quad-Word Memory

To permit wider memory transfers, the Wide Reduceron uses *quad-word* memories allowing any four consecutive locations to be read or written in a single clock cycle. This is *not* the same as saying that memory locations store 72 bits rather than 18 – that would imply that only blocks of words beginning at a four word boundary could be accessed in one cycle. A 72 bit wide memory is easier to build on the Virtex-II, but a quad-word memory facilitates implementation of graph reduction since word alignment issues can be ignored. As the method to implement quad-word memories on FPGA is neither standard nor obvious, and they cannot be synthesised automatically by existing FPGA design tools, we give details. Quad-word memories are built out of four separate memories. If each

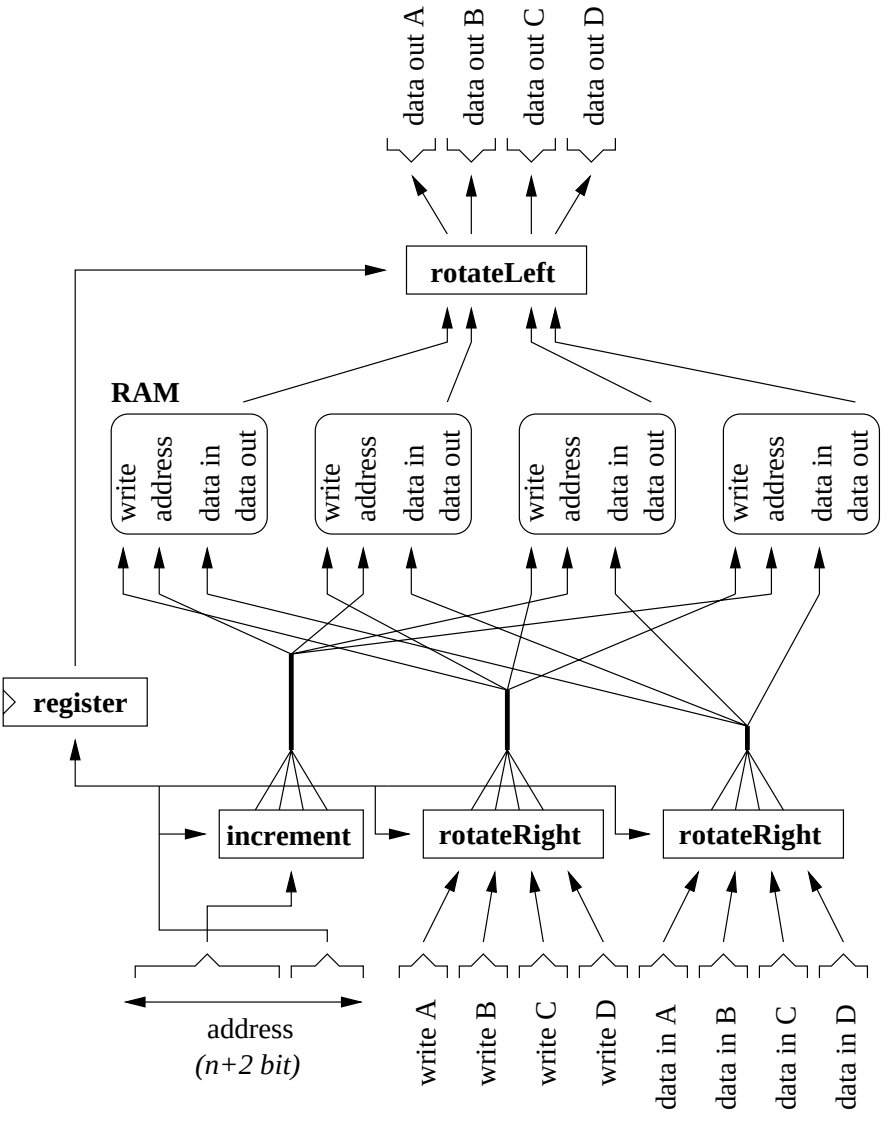


Fig. 5. Circuit diagram for a quad-word memory

internal memory is numbered i where i is drawn from $[0, 1, 2, 3]$ then memory i is used to store locations $[i, i+4, i+8, \ldots]$ of the quad-word memory. Accessing four consecutive locations beginning at an address a is then straightforward if a is a multiple of four, but awkward if it is not. Awkward, but not impossible, because each of the four consecutive locations, beginning at *any* address, *must* be stored in a different internal memory. The problem is then one of rotating

Table 1. The parallel, dual-port, quad-word memories of the Wide Reduceron

Memory	**Capacity** (words)
Combinator (program) store	4k
Heap	32k
Node stack	4k
Address stack	4k
Garbage collector scratch-pad	12k

the quad-word input and output data busses so they line up with those of the internal memories. See Figure 5. The *rotateLeft* and *rotateRight* circuits rotate a given list of inputs by a given number of positions. The *increment* circuit takes an address a and a number n, and produces 4 copies of a, the first n of which are incremented by 1.

Finally, the Wide Reduceron uses quad-word memories that are also *dual-port*, so up to eight words to be accessed together.

4.4 Parallel Memories

As well as widening memory, our semantics also suggested that parallel memories are beneficial, allowing, for example, the stack, heap and combinators to be accessed at the same time during function unfolding. For this reason, the Wide Reduceron has five separate memories, each of which is shown in Table 1 alongside its capacity.

4.5 Garbage Collection

For any serious computations to be performed in such a small amount of memory, a garbage collector is essential. Both versions of the Reduceron use a simple stop-and-copy two-space garbage collector [5]. In this algorithm, active nodes in the heap are copied onto an empty scratch-pad. The scratch-pad, which then contains a compacted copy of the heap, is copied back to the heap again before reduction continues. Although not the cleverest collector, it has the advantage of being extremely simple. Furthermore, the algorithm is easily defined to be *iterative* so no recursive call stack is needed. Our focus is on optimising the reduction process rather than exploring advanced garbage collectors.

4.6 Clock-Level Timing Breakdown

Table 2 shows the number of clock cycles taken to execute each transition rule of the Reduceron. The variable n represents the number of nodes in the body of the supercombinator being unfolded. Each transition rule requires at least two clock cycles because memory is buffered to shorten the critical path (see section 4.2).

Unwinding an application always takes two clock cycles as applications are limited to be a maximum of eight nodes long. The compiler hides this limitation

Table 2. Clock cycles taken by each Reduceron instruction

Operation	Clock cycles
Swap	2
Primitive	3
Unwind	2
Unfold	$3 + \lfloor \frac{n}{8} \rfloor$

by splitting large applications into smaller, nested ones – e.g. $f\ a\ b\ c\ d$ is equivalent to $(f\ a\ b)\ c\ d$.

Another restriction of the implementation is that functions have a maximum of eight parameters. This is because only eight elements of the stack can be accessed simultaneously while instantiating a function body. Again, the limitation can be hidden by the compiler, though our current implementation does not yet do so.

4.7 Description Language

Both versions of the Reduceron are implemented in Haskell using the Lava library [4]. Lava allows circuits to be described by normal Haskell functions over structures of bits (booleans), and can turn such functions into VHDL netlists of FPGA components that can be synthesised by the Xilinx tool set.

We view the *description* of the Reduceron circuit as an interesting aspect of our work. Lava's functional approach has been found to be suitable, despite the Reduceron being an irregular, stateful circuit, non-typical of many Lava applications found in the literature. In particular, we were surprised by how much of the Reduceron could actually be described by *pure* (non-monadic) Lava functions. For example, pure functions are all that are needed to describe the circuit in Figure 5 and the *inst* function in Figure 4. To express the stateful aspects of the circuit we developed a register-transfer monad, similar to the Recipe monad we define in [11]. Having pure functions as the *default* description method is quite appealing: pure functions are typically easy to test and verify; they result in concise, highly parameterised descriptions; and they naturally express circuit parallelism. Only when one needs to express intricate control flow and timing is monadic (sequential) code required.

Unfortunately, although the Reduceron descriptions are quite short, space does not permit presenting them in this paper.

4.8 Resource Usage

The results of synthesising each version of the Reduceron for the Virtex-II (`XC2V2000-6BF957`) using Xilinx ISE 9.1 are shown in Table 3. Concerning clock frequency: a small, carefully optimised 8-bit processor designed by Xilinx (the PicoBlaze) can be clocked at 173.6 MHz on the same device. For the Reduceron to be clocking within a factor of two is acceptable, but suggests room for improvement. One problem with our tool flow is that there is no traceability from

Table 3. Reduceron synthesis results on the Virtex-II (`XC2V2000-6BF957`)

	Baseline	**Wide**	**Maximum**
Number of slices	1530	4874	10752
Number of block RAMs	48	56	56
Clock frequency	94.7 MHz	91.5 MHz	(see text)

code written in Lava to the generated netlist, so it is hard to identify the critical path in the Lava program.

5 Performance

In this section, the impact of the wide memory optimisations is measured by comparing the Baseline and Wide Reducerons running a range of Haskell programs. In addition, the *potential* for special-purpose graph reduction machines is explored by running the same programs using several Haskell implementations on a Pentium-4 2.8GHz PC. The PC Haskell implementations are: Hugs (version May 2006), GHCi (version 6.6), Yhc (latest), Nhc98 (version 1.20), a C implementation of the Reduceron, and the GHC native code compiler (version 6.6) with and without optimisations.

5.1 Programs

Due to the restrictions on the Reduceron, the Haskell programs used in our experiments must: (1) have a maximum heap residency and stack size less than 32k words and 4k words respectively; (2) not take any external input; and (3) produce a single integer as a result. The programs used are:

1. **OrdList.** A program to check the property that insertion into a list preserves ordering for all boolean lists of depth n, applied to $n = 11$.
2. **Perm.** A program to find the smallest number in a list of numbers using a permutation sort, applied to the list containing the numbers 9 down to 1.
3. **MSS.** A program to compute the maximum segment sum of a list of integers applied to the list `[-150..150]`.
4. **Queens.** A function to compute the number of queens that can be placed on an n-by-n chess board such that no queen attacks any other queen, applied to $n = 10$.
5. **Adjoxo.** An adjudicator for noughts and crosses that determines if one side can force victory given a partially complete board. The adjudicator is applied to the empty board.
6. **SumPuz.** A solver for general cryptarithmetic problems. It is applied to a range of problems and outputs the total number of solutions to all of them. (Integer division is not supported on the Reduceron so is implemented by repeated subtraction.)
7. **Sem.** A structural operational semantics of the *While* language [12] applied to a program that naively computes the number of divisors of 1000. (Divisor testing is implemented by repeated subtraction.)

Table 4. Timings of a range programs running on various Haskell implementations

	OrdList	Perm	MSS	Queens	Adjoxo	SumPuz	Sem
Hugs	3.68s	2.70s	3.85s	6.50s	14.81s	4.11s	5.49s
Baseline Red.	13.19s	4.15s	7.41s	7.65s	15.92s	8.98s	15.87s
GHCi	4.26s	2.42s	3.24s	6.35s	7.09s	3.39s	5.36s
Yhc	3.59s	1.76s	1.22s	3.06s	3.85s	2.51s	3.81s
PC Red.	3.65s	1.16s	1.96s	2.33s	5.00s	2.77s	4.50s
Nhc98	3.60s	1.46s	1.38s	2.32s	3.12s	2.28s	3.21s
Wide Red.	1.88s	0.58s	2.01s	1.57s	2.70s	1.67s	2.04s
GHC	0.71s	0.28s	0.38s	0.66s	0.86s	0.47s	0.41s
GHC -O2	0.57s	0.19s	0.28s	0.09s	0.30s	0.27s	0.34s

Table 5. Profiles of programs running on the Wide Reduceron

	OrdList	Perm	MSS	Queens	Adjoxo	SumPuz	Sem
Unwind	31.4%	33.0%	41.7%	32.1%	37.7%	36.8%	28.9%
Unfold	64.0%	54.7%	24.1%	27.8%	37.8%	37.2%	55.8%
Swap	0.0%	4.7%	5.1%	10.8%	7.8%	6.0%	5.8%
Prim.	0.0%	3.5%	7.6%	12.2%	6.8%	5.2%	4.6%
GC	4.6%	4.1%	21.5%	17.0%	9.9%	14.8%	4.9%

5.2 Observations

See tables 4 and 5 for run times and instruction profiles. On average, the Wide Reduceron outperforms the Baseline Reduceron by a factor of six. On heavily arithmetic programs (Queens and MSS) the factor is between three and five, whereas on heavily applicative programs (OrdList and Sem) it is between seven and eight. Unfolding benefits most from wider memory. The average factor of six improvement is significant, but we might have hoped for more considering that eight consecutive locations can be accessed together on each of the five parallel memories. Some suggestions to utilise the parallel memory more fully are given in section 5.4.

On average, the Wide Reduceron (on FPGA) outperforms the Reduceron, Yhc, and Nhc98 bytecode interpreters (on PC). All of these implementations share a common frontend, so each interpreter runs the same core Haskell programs. One of the potential advantages of a bytecode interpreter is that the bytecode can be made sufficiently abstract to have a concise formal semantics, offering hope for a mechanically verified Haskell implementation. However, there is a tension between defining a simple, high-level bytecode and one that is similar enough to the target machine so as to be *efficient*. The Reduceron approach appears to relax this tension; a simple bytecode can be designed without concern for the target machine, and then a machine can be designed to efficiently execute this bytecode. Interestingly, the PC version of the Reduceron performs surprisingly well in comparison to Yhc and Nhc98, considering that it is based

on template instantiation and that Yhc and Nhc98 are G-machine variants. This reinforces the findings of Jansen [9].

The leading native-code compiler GHC performs many advanced optimisations. For example, GHC spots that the critical `safe` function in Queens is strict, so need not be instantiated on the heap. Similar optimisations might be used in a future Reduceron implementation, but architectural changes would be required, e.g. moving from a template instantiation evaluator to an instruction sequence approach. Excluding Queens and Adjoxo, which both involve significant integer operations in a critical loop, and which GHC's optimisations speed up by over a factor of two, the Reduceron (on FPGA) runs, on average, 4.85 times slower than GHC -O2 (on PC).

5.3 Increasing Memory Capacity and Clock Frequency

One of the main limitations of the Reduceron is that it only has 32k words of heap space. This is enough to make an interesting experiment, but too small for any serious application. However, the limitation might be overcome with improved hardware, *without* affecting the existing design significantly. For example, the Computer Architecture group at York have built the PRESENCE-3 FPGA board [13] containing a Virtex-5 FPGA and five large, fast RAMs. Since these RAMs are all accessible in parallel, a wide heap could be obtained using off-chip storage. Further, the Virtex-5 would offer many more block RAMs, permitting larger stack and combinator memories on-chip and therefore to be accessed in parallel as in the existing design.

Another benefit of the Virtex-5 over the Virtex-II is higher performance. The Xilinx synthesis tool states that our current Reduceron design will run at 160 MHz on the Virtex-5. Identifying and reducing the critical path would yield further improvements.

5.4 Possible Design Improvements

Currently, the compiler does not attempt to modify the program to take advantage of the Wide Reduceron's features. In particular, lambda lifting after encoding data types as functions usually introduces a new function definition for each case alternative, breaking function bodies into smaller pieces. While this is desirable for the PC and Baseline Reducerons, larger function bodies should play to the strengths of the Wide Reduceron, and might justify direct support for case expressions and lambda abstractions.

Another limiting factor for memory utilisation is that application nodes are typically only one to five words in size. In particular, the indirections used to achieve sharing are only one node wide. Possible solutions include building combinator spines directly on top of redex roots, and the use of one-level deep trees instead of flat sequences for representing applications.

Eventually, multiple Reducerons could be put on a single FPGA to perform parallel evaluation [7]. The hope is that the flexibility of the FPGA would allow

for a simple yet effective means of parallel reduction. Another avenue of exploration would be to develop a variant of the `ByteString` library [3] which exploits wide memories and vectorised processing logic on the FPGA.

6 Related Work

In the FPCA series of international conferences held between 1981 and 1995, several papers presented designs of exotic new machines to execute functional programs efficiently. Some special-purpose, sequential graph reduction machines were indeed built, including SKIM [16] and NORMA [15]. Unfortunately, at the time, building such machines was a slow and expensive process, and any performance benefit obtained was nullified by the next advancement in stock hardware. Nowadays, the situation is different: FPGA technology has significantly reduced the time and expense required to build custom hardware, and is a widespread, advancing technology in its own right. Furthermore, it appears that users of stock hardware can no longer expect automatic advances in sequential computing speed. Another difference compared with the Reduceron is that both SKIM and NORMA were based on Turner's combinators and did not attempt to use wide, parallel memories to increase performance.

A piece of work similar in spirit to the Reduceron is Augustsson's Big Word Machine (BWM) [1], although the two have been independently. The BWM is a graph reduction machine with a wide word size, specifically four pointers long, allowing wide applications to be quickly built on, and fetched from, the heap. Like the Reduceron, the BWM has a crossbar switch attached to the stack allowing complex rearrangements to be done in a single clock cycle. The BWM also encodes constructors and case expressions using functions and applications respectively. Unlike the Reduceron, the BWM works on an explicit, sequential instruction stream rather than by template instantiation, and it avoids updating the heap in some cases where a computation cannot be shared, thus saving unnecessary heap accesses. Features of the Reduceron not present in the BWM include (1) separate code and heap memories; (2) machine integer support; (3) less memory wastage as data need not be aligned on four-pointer boundaries; and (4) support for building multiple different function applications on the heap simultaneously. The BWM was never actually built. Some simulations were performed but Augustsson writes "the absolute performance of the machine is hard to determine at this point" [1].

7 Conclusion

In the introduction we argued that the von Neumann bottleneck impedes the performance of graph reduction on standard computers, and suggested that the problem could be overcome by building a special-purpose machine with wide, parallel memory units. We have explored this very possibility by building a prototype of such a machine – the Reduceron – using an FPGA. The combination of wide, parallel memory units and vectorised processing logic on the Reduceron

gives a factor of six speed-up on average across a range of benchmark programs. Furthermore, running at 91.5MHz on a Xilinx Virtex-II FPGA, the Reduceron performs better than interpreted bytecode and often within a small factor of optimised native-code running on a 2.8GHz Pentium-4 PC. Considering the large performance advantage of conventional *hard* processors over *soft*, FPGA-based ones for executing C programs [10], and the *simplicity* of the Reduceron, it would certainly be an interesting result if, after further work on the Reduceron, Haskell programs were found to run at comparable speeds on both.

FPGAs have, to a large extent, eliminated the effort and expertise needed to build custom hardware. They may be viewed as an advancing technology that continues to offer higher performance, perhaps one day approaching the clock rates of modern PCs. Or, alternatively, as a tool for rapidly prototyping designs before they are manufactured as efficient, non-programmable ASICs. Both views, along with the results obtained in this paper, motivate further experiments in the design of special-purpose graph reduction machines using FPGAs. The hope is that researchers can find *simple* and *elegant* yet *fast* and *parallel* reduction methods by side-stepping the constraints and intricacies of standard, von Neumann, computers.

Acknowledgements

The first author is supported by an award from the Engineering and Physical Sciences Research Council of the United Kingdom. We thank Jack Whitham and Ian Gray for their help in the remote lab and the Real Time Systems group for providing the Virtex-II FPGA. We also thank Emil Axelsson for comments on a draft, and the anonymous IFL reviewers for both encouraging and critical comments! Most of all, we thank Neil Mitchell for many useful discussions, for making Yhc Core such a pleasure to use, and for providing several transformations including the data type encoder and the lambda lifter.

References

1. Augustsson, L.: BWM: A Concrete Machine for Graph Reduction. In: Proceedings of the 1991 Glasgow Workshop on Functional Programming, London, UK, pp. 36–50. Springer, Heidelberg (1992)
2. Burn, G.L., Peyton Jones, S.L., Robson, J.D.: The spineless G-machine. In: LFP 1988: Proceedings of the 1988 ACM conference on LISP and functional programming, pp. 244–258. ACM, New York (1988)
3. Coutts, D., Stewart, D., Leshchinskiy, R.: Rewriting haskell strings. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 50–64. Springer, Heidelberg (2006)
4. Claessen, K.: Embedded Languages for Describing and Verifying Hardware. PhD Thesis, Chalmers University of Technology (2001)
5. Fenichel, R.R., Yochelson, J.C.: A LISP garbage-collector for virtual-memory computer systems. Commun. ACM 12(11), 611–612 (1969)
6. Golubovsky, D., Mitchell, N., Naylor, M.: Yhc.Core - from Haskell to Core. The Monad.Reader (7), 45–61 (2007)

7. Hammond, K., Michelson, G. (eds.): Research Directions in Parallel Functional Programming. Springer, London (2000)
8. Hughes, R.J.M.: Super Combinators—A New Implementation Method for Applicative Languages. In: Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, pp. 1–10 (1982)
9. Jansen, J.M., Koopman, P., Plasmeijer, R.: Efficient interpretation by transforming data types and patterns to functions. In: Trends in Functional Programming, vol. 7, Intellect (2007)
10. Lysecky, R., Vahid, F.: A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In: DATE 2005: Proceedings of the conference on Design, Automation and Test in Europe, Washington, DC, USA, pp. 18–23. IEEE Computer Society, Los Alamitos (2005)
11. Naylor, M.: A Recipe for Controlling Lego using Lava. The Monad.Reader (7), 5–21 (2007)
12. Riis Nielson, H., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley, Chichester (1992)
13. University of York. PRESENCE-3 Development, `http://www.cs.york.ac.uk/arch/neural/hardware/presence-3/`
14. Peyton Jones, S.: The Implementation of Functional Programming Languages. Computer Science. Prentice-Hall, Englewood Cliffs (1987)
15. Scheevel, M.: NORMA: a graph reduction processor. In: LFP 1986: Proceedings of the 1986 ACM conference on LISP and functional programming, pp. 212–219. ACM, New York (1986)
16. Stoye, W.: The Implementation of Functional Languages using Custom Hardware. PhD Thesis, University of Cambridge (1985)
17. The Yhc Team. The York Haskell Compiler - user's guide (February 2007), `http://www.haskell.org/haskellwiki/Yhc`

A Supercompiler for Core Haskell

Neil Mitchell and Colin Runciman

University of York, UK
http://www.cs.york.ac.uk/~ndm

Abstract. Haskell is a functional language, with features such as higher order functions and lazy evaluation, which allow succinct programs. These high-level features present many challenges for optimising compilers. We report practical experiments using novel variants of *supercompilation*, with special attention to let bindings and the generalisation technique.

1 Introduction

Haskell [17] can be used in a highly declarative manner, to express specifications which are themselves executable. Take for example the task of counting the number of words in a file read from the standard input. In Haskell, one could write:

$$\mathsf{main} = \mathsf{print} \circ \mathsf{length} \circ \mathsf{words} =\!\ll \mathsf{getContents}$$

From right to left, the getContents function reads the input as a list of characters, words splits this list into a list of words, length counts the number of words, and finally print writes the value to the screen.

An equivalent C program is given in Figure 1. Compared to the C program, the Haskell version is more concise and more easily seen to be correct. Unfortunately, the Haskell program (compiled with GHC [25]) is also three times slower than the C version (compiled with GCC). This slowdown is caused by several factors:

Intermediate Lists. The Haskell program produces and consumes many intermediate lists as it computes the result. The getContents function produces a list of characters, words consumes this list and produces a list of lists of characters, length then consumes the outermost list. The C version uses no intermediate data structures.

Functional Arguments. The words function is defined using the dropWhile function, which takes a predicate and discards elements from the input list until the predicate becomes true. The predicate is passed as an invariant function argument in all applications of dropWhile.

Laziness and Thunks. The Haskell program proceeds in a lazy manner, first demanding one character from getContents, then processing it with each of the functions in the pipeline. At each stage, a lazy thunk for the remainder of each function is created.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 147–164, 2008.

```
int main()
{
        int i = 0;
        int c, last_space = 1, this_space;
        while ((c = getchar()) != EOF) {
                this_space = isspace(c);
                if (last_space && !this_space)
                        i++;
                last_space = this_space;
        }
        printf("%i\n", i);
        return 0;
}
```

Fig. 1. Word counting in C

Using the optimiser developed in this paper, named Supero, we can eliminate all these overheads. We obtain a program that performs *faster* than the C version. The optimiser is based around the techniques of supercompilation [29], where some of the program is evaluated at compile time, leaving an optimised residual program.

Our goal is an automatic optimisation that makes high-level Haskell programs run as fast as low-level equivalents, eliminating the current need for hand-tuning and low-level techniques to obtain competitive performance. We require no annotations on any part of the program, including the library functions.

1.1 Contributions

- To our knowledge, this is the first time supercompilation has been applied to Haskell.
- We make careful study of the let expression, something absent from the Core language of many other papers on supercompilation.
- We present an alternative generalisation step, based on a homeomorphic embedding [9].

1.2 Roadmap

We first introduce a Core language in §2, on which all transformations are applied. Next we describe our supercompilation method in §3. We then give a number of benchmarks, comparing both against C (compiled with GCC) in §4 and Haskell (compiled with GHC) in §5. Finally, we review related work in §6 and conclude in §7.

2 Core Language

Our supercompiler uses the Yhc-Core language [6]. The expression type is given in Figure 2. A program is a mapping of function names to expressions. Our

$$
\begin{array}{lll}
\mathsf{expr} = v & & \text{variable} \\
\quad \mid c & & \text{constructor} \\
\quad \mid f & & \text{function} \\
\quad \mid x\ \overline{ys} & & \text{application} \\
\quad \mid \lambda\overline{vs} \to x & & \text{lambda abstraction} \\
\quad \mid \textbf{let}\ v = x\ \textbf{in}\ y & & \text{let binding} \\
\quad \mid \textbf{case}\ x\ \textbf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\} & & \text{case expression} \\
\\
\mathsf{pat}\ = c\ \overline{vs} & &
\end{array}
$$

Where v ranges over variables, c ranges over constructors, f ranges over functions, x and y range over expressions and p ranges over patterns.

Fig. 2. Core syntax

$$
\begin{array}{rl}
\mathsf{split}\ (v) & = (v, [\,]) \\
\mathsf{split}\ (c) & = (c, [\,]) \\
\mathsf{split}\ (f) & = (f, [\,]) \\
\mathsf{split}\ (x\ \overline{ys}) & = (\bullet\ \overline{\bullet}, x : \overline{ys}) \\
\mathsf{split}\ (\lambda\overline{vs} \to x) & = (\lambda\overline{vs} \to \bullet, x) \\
\mathsf{split}\ (\textbf{let}\ v = x\ \textbf{in}\ y) & = (\textbf{let}\ v = \bullet\ \textbf{in}\ \bullet\,, [x, y]) \\
\mathsf{split}\ (\textbf{case}\ x\ \textbf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\}) & = (\textbf{case}\ \bullet\ \textbf{of}\ \{p_1 \to \bullet; ...; p_n \to \bullet\}, [x, y_1, ..., y_n])
\end{array}
$$

Fig. 3. The split function, returning a spine and all subexpressions

Core language is higher order and lazy, but lacks much of the syntactic sugar found in Haskell. Pattern matching occurs only in case expressions, and all case expressions are exhaustive. All names are fully qualified. Haskell's type classes have been removed using the dictionary transformation [32].

The Yhc compiler, a fork of nhc [22], can output Core files. Yhc can also link in all definitions from all required libraries, producing a single Core file representing a whole program.

The primary difference between Yhc-Core and GHC-Core [26] is that Yhc-Core is untyped. The Core is generated from well-typed Haskell, and is guaranteed not to fail with a type error. All the transformations could be implemented equally well in a typed Core language, but we prefer to work in an untyped language for simplicity of implementation.

In order to avoid accidental variable name clashes while performing transformations, we demand that all variables within a program are unique. All transformations may assume this invariant, and must maintain it.

We define the split function in Figure 3, which splits an expression into a pair of its spine and its immediate subexpressions. The • markers in the spine indicate the positions from which subexpressions have been removed. We define the join operation to be the inverse of split, taking a spine and a list of expressions, and producing an expression.

```
supercompile ()
  seen := { }
  bind := { }
  tie ({ }, main)

tie (ρ, x)
  if x ∉ seen then
    seen := seen ∪ { x }
    bind := bind ∪ { ψ(x) = λfv(x) → drive(ρ, x) }
  endif
  return (ψ(x) fv(x))

drive (ρ, x)
  if terminate(ρ, x) then
    (a, b) = split(generalise(x))
    return join(a, map (tie ρ) b)
  else
    return drive(ρ ∪ { x }, unfold (x))
```

Where ψ is a mapping from expressions to function names, and $\mathsf{fv}(x)$ returns the free variables in x. This code is parameterised by: terminate which decides whether to stop supercompilation of this expression; generalise which generalises an expression before residuation; unfold which chooses a function application and unfolds it.

Fig. 4. The supercompile function

3 Supercompilation

Our supercompiler takes a Core program as input, and produces an equivalent Core program as output. To improve the program we do not make small local changes to the original, but instead *evaluate it* so far as possible at compile time, leaving a *residual program* to be run.

The general method of supercompilation is shown in Figure 4. Each function in the output program is an optimised version of some associated expression in the input program. Supercompilation starts at the main function, and supercompiles the expression associated with main. Once the expression has been supercompiled, the outermost element in the expression becomes part of the residual program. All the subexpressions are assigned names, and will be given definitions in the residual program. If any expression (up to alpha renaming) already has a name in the residual program, then the same name is used. Each of these named inner expressions are then supercompiled as before.

The supercompilation of an expression proceeds by repeatedly inlining a function application until some termination criterion is met. Once the termination criterion holds, the expression is generalised before the outer spine becomes part of the residual program and all immediate subexpressions are assigned names. After each inlining step, the expression is simplified using the rules in Figure 5. There are three key decisions in the supercompilation of an expression:

1. Which function to inline.
2. What termination criterion to use.
3. What generalisation to use.

The original Supero work [13] inlined following evaluation order (with the exception of let expressions), used a bound on the size of the expression to

$$\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\})\ \textbf{of}\ \overline{alts}$$
$$\Rightarrow \textbf{case}\ x\ \textbf{of}\ \{p_1 \to \textbf{case}\ y_1\ \textbf{of}\ \overline{alts}$$
$$; ...$$
$$; p_n \to \textbf{case}\ y_n\ \textbf{of}\ \overline{alts}\}$$

$$\textbf{case}\ c\ x_1 ... x_n\ \textbf{of}\ \{...; c\ v_1 ... v_n \to y; ...\}$$
$$\Rightarrow \textbf{let}\ v_1 = x_1\ \textbf{in}$$
$$...$$
$$\textbf{let}\ v_n = x_n\ \textbf{in}$$
$$y$$

$$\textbf{case}\ v\ \textbf{of}\ \{...; c\ \overline{vs} \to x; ...\}$$
$$\Rightarrow \textbf{case}\ v\ \textbf{of}\ \{...; c\ \overline{vs} \to x\ [v\ /\ c\ \overline{vs}]; ...\}$$

$$\textbf{case}\ (\textbf{let}\ v = x\ \textbf{in}\ y)\ \textbf{of}\ \overline{alts}$$
$$\Rightarrow \textbf{let}\ v = x\ \textbf{in case}\ y\ \textbf{of}\ \overline{alts}$$

$$(\textbf{let}\ v = x\ \textbf{in}\ y)\ z$$
$$\Rightarrow \textbf{let}\ v = x\ \textbf{in}\ y\ z$$

$$(\textbf{case}\ x\ \textbf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\})\ z$$
$$\Rightarrow \textbf{case}\ x\ \textbf{of}\ \{p_1 \to y_1\ z; ...; p_n \to y_n\ z\}$$

$$(\lambda v \to x)\ y$$
$$\Rightarrow \textbf{let}\ v = y\ \textbf{in}\ x$$

$$\textbf{let}\ v = x\ \textbf{in}\ (\textbf{case}\ y\ \textbf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\})$$
$$\Rightarrow \textbf{case}\ y\ \textbf{of}\ \{p_1 \to \textbf{let}\ v = x\ \textbf{in}\ y_1$$
$$; ...$$
$$; p_n \to \textbf{let}\ v = x\ \textbf{in}\ y_n\}$$
$$\textbf{where}\ v \text{ is not used in } y$$

$$\textbf{let}\ v = x\ \textbf{in}\ y$$
$$\Rightarrow y\ [v\ /\ x]$$
$$\textbf{where}\ x \text{ is a lambda, a variable, or } v \text{ is used once in } y$$

$$\textbf{let}\ v = c\ x_1 ... x_n\ \textbf{in}\ y$$
$$\Rightarrow \textbf{let}\ v_1 = x_1\ \textbf{in}$$
$$...$$
$$\textbf{let}\ v_n = x_n\ \textbf{in}$$
$$y\ [v\ /\ c\ v_1 ... v_n]$$
$$\textbf{where}\ v_1 ... v_n \text{ are fresh}$$

Fig. 5. Simplification rules

ensure termination, and performed no generalisation. First we give examples of our supercompiler in use, then we return to examine each of the three choices we have made.

3.1 Examples of Supercompilation

Example 1: Supercompiling and Specialisation

$$\mathsf{main}\ as = \mathsf{map}\ (\lambda b \to b{+}1)\ as$$

$$\begin{aligned}\mathsf{map}\ f\ cs = \textbf{case}\ cs\ \textbf{of}\\ [\,] \quad &\to [\,]\\ d : ds &\to f\ d : \mathsf{map}\ f\ ds\end{aligned}$$

There are two primary inefficiencies in this example: (1) the map function passes the f argument invariantly in every call; (2) the application of f is more expensive than if the function was known in advance.

The supercompilation proceeds by first assigning a new unique name (we choose h_0) to $\mathsf{map}\ (\lambda b \to b{+}1)\ as$, providing parameters for each of the free variables in the expression, namely as. We then choose to expand map, and invoke the simplification rules:

$$\begin{aligned}h_0\ as &= \mathsf{map}\ (\lambda b \to b{+}1)\ as\\ &= \textbf{case}\ as\ \textbf{of}\\ &\qquad [\,] \quad \to [\,]\\ &\qquad d : ds \to d{+}1 : \mathsf{map}\ (\lambda b \to b{+}1)\ ds\end{aligned}$$

We now have a **case** with a variable as the scrutinee at the root of the expression, which cannot be reduced further, so we residuate the spine. When processing the expression $\mathsf{map}\ (\lambda b \to b{+}1)\ ds$ we spot this to be an alpha renaming of the body of an existing generated function, namely h_0, and use this function:

$$\begin{aligned}h_0\ as = \textbf{case}\ as\ \textbf{of}\\ [\,] \quad &\to [\,]\\ d : ds &\to d{+}1 : h_0\ ds\end{aligned}$$

We have now specialised the higher-order argument, passing less data at runtime. □

Example 2: Supercompiling and Deforestation
The deforestation transformation [31] removes intermediate lists from a traversal. A similar result is obtained by applying supercompilation, as shown here. Consider the operation of mapping $(*2)$ over a list and then mapping $(+1)$ over the result. The first map deconstructs one list, and constructs another. The second does the same.

$$\mathsf{main}\ as = \mathsf{map}\ (\lambda b \to b{+}1)\ (\mathsf{map}\ (\lambda c \to c{*}2)\ as)$$

We first assign a new name for the body of main, then choose to expand the outer call to map:

$$
\begin{array}{l}
h_0\ as = \textbf{case}\ \mathsf{map}\ (\lambda c \to c{*}2)\ as\ \textbf{of} \\
\qquad\qquad [\,] \quad\ \to [\,] \\
\qquad\qquad d : ds \to d{+}1 : \mathsf{map}\ (\lambda b \to b{+}1)\ ds
\end{array}
$$

Next we choose to inline the map scrutinised by the case, then perform the **case**/**case** simplification, and finally residuate:

$$
\begin{array}{l}
h_0\ as = \textbf{case}\ (\textbf{case}\ as\ \textbf{of} \\
\qquad\qquad\qquad [\,] \quad\ \to [\,] \\
\qquad\qquad\qquad e : es \to e{*}2 : \mathsf{map}\ (\lambda c \to c{*}2)\ es)\ \textbf{of} \\
\qquad\qquad [\,] \quad\ \to [\,] \\
\qquad\qquad d : ds \to y{+}1 : \mathsf{map}\ (\lambda b \to b{+}1)\ ds \\
\\
\qquad = \textbf{case}\ as\ \textbf{of} \\
\qquad\qquad [\,] \quad\ \to [\,] \\
\qquad\qquad d : ds \to (y{*}2){+}1 : \mathsf{map}\ (\lambda b \to b{+}1)\ (\mathsf{map}\ (\lambda c \to c{*}2)\ ds) \\
\\
\qquad = \textbf{case}\ as\ \textbf{of} \\
\qquad\qquad [\,] \quad\ \to [\,] \\
\qquad\qquad d : ds \to (y{*}2){+}1 : h_0\ ds
\end{array}
$$

Both intermediate lists have been removed, and the functional arguments to map have both been specialised. □

3.2 Which Function to Inline

During the supercompilation of an expression, at each step some function needs to be inlined. Which to choose? In most supercompilation work the choice is made following the runtime semantics of the program. But in a language with let expressions this may be inappropriate. If a function in a let binding is inlined, its application when reduced may be simple enough to substitute in the let body. However, if a function in a let body is inlined, the let body may now only refer to the let binding once, allowing the binding to be substituted. Let us take two expressions, based on intermediate steps obtained from real programs (word counting and prime number calculation respectively):

$$
\begin{array}{ll}
\textbf{let}\ x = (\equiv)\ \$\ 1 & \textbf{let}\ x = \mathsf{repeat}\ 1 \\
\textbf{in}\ x\ 1 : \mathsf{map}\ x\ ys & \textbf{in}\ \mathsf{const}\ 0\ x : \mathsf{map}\ f\ x
\end{array}
$$

In the first example, inlining ($) in the let binding gives $(\lambda x \to 1 \equiv x)$, which is now simple enough to substitute for x, resulting in $(1 \equiv 1 : \mathsf{map}\ (\lambda x \to 1 \equiv x)\ ys)$ after simplification. Now map can be specialised appropriately. Alternatively, expanding the map repeatedly would keep increasing the size of expression until the termination criterion was met, aborting the supercompilation of this expression without achieving specialisation.

Taking the second example, repeat can be inlined indefinitely. However, by unfolding the const we produce $\textbf{let}\ x = \mathsf{repeat}\ 1\ \textbf{in}\ 0 : \mathsf{map}\ f\ x$. Since x is

only used once we substitute it to produce $(0 : \mathsf{map}\ f\ (\mathsf{repeat}\ 1))$, which can be deforested.

Unfortunately these two examples seem to suggest different strategies for unfolding – unfold in the let binding or unfold in the let body. However, they do have a common theme – unfold the function that cannot be unfolded infinitely often. Our strategy can be defined by the unfold function:

```
unfold x = head (filter (not ∘ terminate) xs ++ xs ++ [x])
  where xs = unfolds x

unfolds f | f is a function = [inline f]
unfolds x = [join spine (sub |+ (i, y))
            | let (spine, sub) = split x
            , i ← [0 .. length sub], y ← unfolds (sub !! i)]
  where xs |+ (i, x) = zipWith (λj y → if i ≡ j then x else y) [0 ..] xs
```

The unfolds function computes all possible one-step inlinings, using an in-order traversal of the abstract syntax tree. The unfold function chooses the first unfolding which does not cause the supercompilation to terminate. If no such expression exists, the first unfolding is chosen.

3.3 The Termination Criterion

The original Supero program used a size bound on the expression to determine when to stop. The problem with a size bound is that different programs require different bounds to ensure both timely completion at compile-time and efficient residual programs. Indeed, within a single program, there may be different elements requiring different size bounds – a problem exacerbated as the size and complexity of a program increases.

We use the termination criterion suggested by Sørensen and Glück [24] – homeomorphic embedding. An expression x is an embedding of y, written $x \trianglelefteq y$, if the relationship can be inferred by the rules:

$$\frac{\mathrm{dive}(x, y)}{x \trianglelefteq y} \qquad \frac{\mathrm{couple}(x, y)}{x \trianglelefteq y}$$

$$\frac{s \trianglelefteq t_i \text{ for some } i}{\mathrm{dive}(s, \sigma(t_1, \ldots, t_n))} \qquad \frac{\sigma_1 \sim \sigma_2, s_1 \trianglelefteq t_1, \ldots, s_n \trianglelefteq t_n}{\mathrm{couple}(\sigma_1(s_1, \ldots, s_n), \sigma_2(t_1, \ldots, t_n))}$$

The homeomorphic embedding uses the relations dive and couple. The dive relation checks if the first term is contained as a child of the second term, while the couple relation checks if both terms have the same outer shell. We use σ to denote the spine of an expression, with $s_1, \ldots, s_n$ being its subexpressions. We test for equivalence of σ_1 and σ_2 using the $\sim$ relation, a weakened form of equality where all variables are considered equal. We terminate the supercompilation of an expression y if on the chain of reductions from main to y we have encountered an expression x such that $x \trianglelefteq y$.

In addition to using the homeomorphic embedding, we also terminate if further unfolding cannot yield any improvement to the root of the expression. For example, if the root of an expression is a constructor application, no further unfolding will change the root constructor. When terminating for this reason, we always residuate the outer spine of the expression, without applying any generalisation.

3.4 Generalisation

When the termination criterion has been met, it is necessary to discard information about the current expression, so that the supercompilation terminates. We always residuate the outer spine of the expression, but first we attempt to generalise the expression so that the information lost is minimal. The paper by Sørensen and Glück provides a method for generalisation, which works by taking the most specific generalisation of the current expression and expression which is a homeomorphic embedding of it.

The most specific generalisation of two expressions s and t, $\text{msg}(s,t)$, is produced by applying the following rewrite rule to the initial triple $(x, \{x = s\}, \{x = t\})$, resulting in a common expression and two sets of bindings.

$$\begin{pmatrix} t_g \\ \{x = \sigma(s_1, \ldots, s_n)\} \cup \theta_1 \\ \{x = \sigma(t_1, \ldots, t_n)\} \cup \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g[x/\sigma(y_1, \ldots, y_n)] \\ \{y_1 = s_1, \ldots, y_n = s_n\} \cup \theta_1 \\ \{y_1 = t_1, \ldots, y_n = t_n\} \cup \theta_2 \end{pmatrix}$$

Our generalisation is characterised by $x \bowtie y$, which produces an expression equivalent to y, but similar in structure to x.

$x \bowtie \sigma^*(y)$, if $\text{dive}(x, \sigma^*(y)) \wedge \text{couple}(x, y)$
let $f = \lambda\overline{vs} \rightarrow x$ **in** $\sigma^*(f\ vs)$
where $\overline{vs} = \text{fv}(y) \backslash \text{fv}(\sigma^*(y))$

$x \bowtie y$, if $\text{couple}(x, y)$
let θ_2 **in** t_g
where $(t_g, \theta_1, \theta_2) = \text{msg}(x, y)$

The fv function in the first rule calculates the free variables of an expression, and $\sigma^*(y)$ denotes a subexpression y within a containing context σ^*. The first rule applies if the homeomorphic embedding first applied the dive rule. The idea is to descend to the element which matched, and then promote this to the top-level using a lambda. The second rule applies the most specific generalisation operation if the coupling rule was applied first. We now show an example where most specific generalisation fails to produce the ideal generalised version.

Example 3

```
case putStr (repeat '1') r of
    (r, _) → (r, ())
```

This expression (which we name x) prints an infinite stream of 1's. The pairs and r's correspond to the implementation of GHC's IO Monad [16]. After several unrollings, we obtain the expression (named x'):

```
case putChar '1' r of
    (r, _) → case putStr (repeat '1') r of
                 (r, _) → (r, ())
```

The homeomorphic embedding $x \trianglelefteq x'$ matches, detecting an occurrence of the **case** putStr ... expression, and the supercompilation of x' is stopped. The most specific generalisation rule is applied as $\text{msg}(x, x')$ and produces:

```
let a = putChar
    b = '1'
    c = λr → case putStr (repeat '1') r of
                 (r, _) → (r, ())
in case a b r of
       (r, _) → c r
```

The problem is that msg works from the top, looking for a common root of both expression trees. However, if the first rule applied by $\trianglelefteq$ was dive, the roots may be unrelated. Using our generalisation, $x \bowtie x'$:

```
let x = λr → case putStr (repeat '1') r of
                 (r, _) → (r, ())
in case putChar '1' r of
       (r, _) → x r
```

Our generalisation is superior because it has split out the putStr application *without* lifting the putChar application or the constant '1'. The putChar application can now be supercompiled further in the context of the case expression. □

4 Performance Compared with C Programs

The benchmarks we have used as motivating examples are inspired by the Unix `wc` command – namely character, word and line counting. We require the program to read from the standard input, and write out the number of elements in the file. To ensure that we test computation speed, not IO speed (which is usually determined by the buffering strategy, rather than optimisation) we demand that all input is read using the standard C getchar function only. Any buffering improvements, such as reading in blocks or memory mapping of files, could be performed equally in all compilers.

All the C versions are implemented following a similar pattern to Figure 1. Characters are read in a loop, with an accumulator recording the current value. Depending on the program, the body of the loop decides when to increment the accumulator. The Haskell versions all follow the same pattern as in the Introduction, merely replacing words with lines, or removing the words function for character counting.

We performed all benchmarks on a machine running Windows XP, with a 3GHz processor and 1Gb RAM. All benchmarks were run over a 50Mb log file,

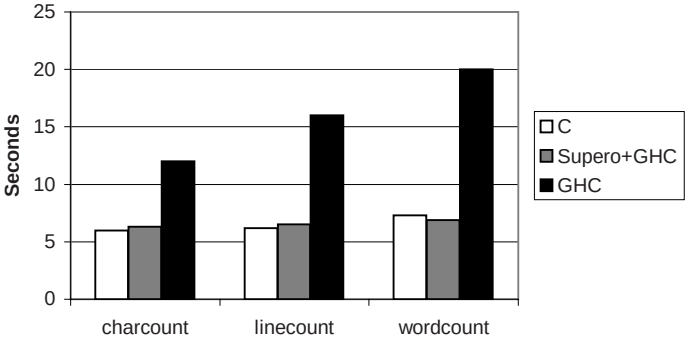


Fig. 6. Benchmarks with C, Supero+GHC and GHC alone

repeated 10 times, and the lowest value was taken. The C versions used GCC[1] version 3.4.2 with -O3. The Haskell version used GHC 6.8.1 with -O2. The Supero version was compiled using our optimiser, then written back as a Haskell file, and compiled once more with GHC 6.8.1 and -O2.

The results are given in Figure 6. In all the benchmarks C and Supero are within 10% of each other, while GHC trails further behind.

4.1 Identified Haskell Speedups

During initial trials using these benchmarks, we identified two unnecessary bottlenecks in the Haskell version of word counting. Both were remedied before the presented results were obtained.

Slow isSpace function. The first issue is that isSpace in Haskell is much more expensive than isspace in C. The simplest solution is to use a FFI (Foreign Function Interface) [16] call to the C isspace function in all cases, removing this factor from the benchmark. A GHC bug (number 1473) has been filed about the slow performance of isSpace.

Inefficient words function. The second issue is that the standard definition of the words function (given in Figure 7) performs two additional isSpace tests per word. By appealing to the definitions of dropWhile and break it is possible to show that in words the first character of x is not a space, and that if y is non-empty then the first character is a space. The revised words′ function uses these facts to avoid the redundant isSpace tests.

4.2 Potential GHC Speedups

We have identified three factors limiting the performance of residual programs when compiled by GHC. These problems cannot be solved at the level of Core

[1] `http://gcc.gnu.org/`

```
words :: String → [String]
words s = case dropWhile isSpace s of
                  [] → []
                  x  → w : words y
                       where (w, y) = break isSpace x

words' s = case dropWhile isSpace s of
                  []     → []
                  x : xs → (x : w) : words' (drop1 z)
                           where (w, z) = break isSpace xs

drop1 []       = []
drop1 (x : xs) = xs
```

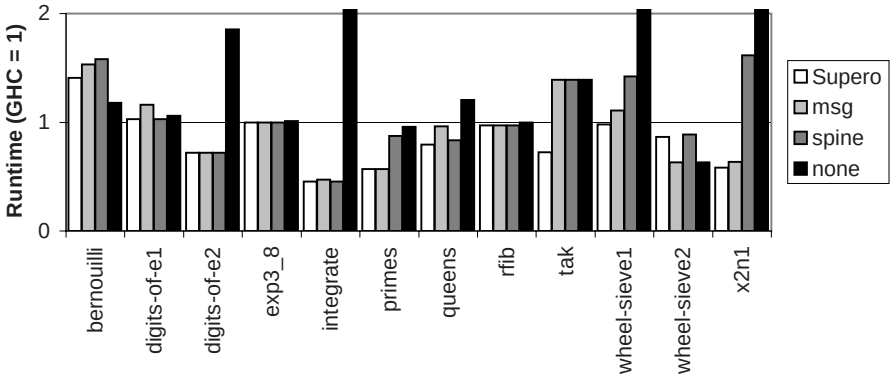
Fig. 7. The words function from the Haskell standard libraries, and an improved words′

transformations. We suspect that by fixing these problems, the Supero execution time would improve by between 5% and 15%.

Strictness inference. The GHC compiler is overly conservative when determining strictness for functions which use the FFI (GHC bug 1592). The getchar function is treated as though it may raise an exception, and terminate the program, so strict arguments are not determined to be strict. If GHC provided some way to mark an FFI function as not generating exceptions, this problem could be solved. The lack of strictness information means that in the line and word counting programs, every time the accumulator is incremented, the number is first unboxed and then reboxed [19].

Heap checks. The GHC compiler follows the standard STG machine [15] design, and inserts heap checks before allocating memory. The purpose of a heap check is to ensure that there is sufficient memory on the heap, so that allocation of memory is a cheap operation guaranteed to succeed. GHC also attempts to lift heap checks: if two branches of a case expression both have heap checks, they are replaced with one shared heap check before the case expression. Unfortunately, with lifted heap checks, a tail-recursive function that allocates memory only upon exit can have the heap test executed on every iteration (GHC bug 1498). This problem affects the character counting example, but if the strictness problems were solved, it would apply equally to all the benchmarks.

Stack checks. The final source of extra computation relative to the C version are stack checks. Before using the stack to store arguments to a function call, a test is performed to check that there is sufficient space on the stack. Unlike the heap checks, it is necessary to analyse a large part of the flow of control to determine when these checks are unnecessary. It is not clear how to reduce stack checks in GHC.



Supero uses the ⋈ generalisation method; **msg** uses the msg function for generalisation; **spine** applies no generalisation operation; **none** never performs any inlining.

Fig. 8. Runtime, relative to GHC being 1

5 Performance Compared with GHC Alone

The standard set of Haskell benchmarks is the nofib suite [14]. It is divided into three categories of increasing size: imaginary, spectral and real. Even small Haskell programs increase in size substantially once libraries are included, so we have limited our attention to the benchmarks in the imaginary section. All benchmarks were run with parameters that require runtimes of between 3 and 5 seconds for GHC.

We exclude two benchmarks, paraffins and gen_regexps. The paraffins benchmark makes substantial use of arrays, and we have not yet mapped the array primitives of Yhc onto those of GHC, which is necessary to run the transformed result. The gen_regexps benchmark tests character processing: for some reason (as yet unknown) the supercompiled executable fails.

The results of these benchmarks are given in Figure 8, along with detailed breakdowns in Table 1. All results are relative to the runtime of a program compiled with GHC -O2, lower numbers being better. The first three variants (Supero, msg, spine) all use homeomorphic embedding as the termination criterion, and ⋈, msg or nothing respectively as the generalisation function. The final variant, none, uses a termination test that always causes a residuation. The 'none' variant is useful as a control to determine which improvements are due to bringing all definitions into one module scope, and which are a result of supercompilation. Compilation times ranged from a few seconds to five minutes.

The Bernouilli benchmark is the only one where Supero is slower than GHC by more than 3%. The reason for this anomaly is that a dictionary is referred to in an inner loop which is specialised away by GHC, but not by Supero.

With the exception of the wheel-sieve2 benchmark, our ⋈ generalisation strategy performs as well as, or better than, the alternatives. While the msg general-

Table 1. Runtime, relative to GHC being 1

Program	Supero	msg	spine	none	Size	Memory
bernouilli	1.41	1.53	1.58	1.18	1.10	0.97
digits-of-e1	1.03	1.16	1.03	1.06	1.01	1.11
digits-of-e2	0.72	0.72	0.72	1.86	1.00	0.84
exp3_8	1.00	1.00	1.00	1.01	0.99	1.00
integrate	0.46	0.47	0.46	4.01	1.02	0.08
primes	0.57	0.57	0.88	0.96	1.00	0.98
queens	0.79	0.96	0.83	1.21	1.01	0.85
rfib	0.97	0.97	0.97	1.00	1.00	1.08
tak	0.72	1.39	1.39	1.39	1.00	1.00
wheel-sieve1	0.98	1.11	1.42	5.23	1.19	2.79
wheel-sieve2	0.87	0.63	0.89	0.63	1.49	2.30
x2n1	0.58	0.64	1.61	3.04	1.09	0.33

Program is the name of the program; **Supero** uses the $\bowtie$ generalisation method; **msg** uses the msg function for generalisation; **spine** applies no generalisation operation; **none** never performs any inlining; **Size** is the size of the Supero generated executable; **Memory** is the amount of memory allocated on the heap by the Supero executable.

isation performs better than the empty generalisation on average, the difference is not as dramatic.

5.1 GHC's Optimisations

For these benchmarks it is important to clarify which optimisations are performed by GHC, and which are performed by Supero. The 'none' results show that, on average, taking the Core output from Yhc and compiling with GHC does *not* perform as well as the original program compiled using GHC. GHC has two special optimisations that work in a restricted number of cases, but which Supero produced Core is unable to take advantage of.

Dictionary Removal. Functions which make use of type classes are given an additional dictionary argument. In practice, GHC specialises many such functions by creating code with a particular dictionary frozen in. This optimisation is specific to type classes – a tuple of higher order functions is not similarly specialised. After compilation with Yhc, the type classes have already been converted to tuples, so Supero must be able to remove the dictionaries itself. One benchmark where dictionary removal is critical is digits-of-e2.

List Fusion. GHC relies on names of functions, particularly foldr/build [21], to apply special optimisation rules such as list fusion. Many of GHC's library functions, for example iterate, are defined in terms of foldr to take advantage of these special properties. After transformation with Yhc, these names are destroyed, so no rule based optimisation can be performed. One example where list fusion is critical is primes, although it occurs in most of the benchmarks to some extent.

6 Related Work

Supercompilation [29,30] was introduced by Turchin for the Refal language [28]. Since this original work, there have been various suggestions of both termination strategies and generalisation strategies [9,24,27]. The original supercompiler maintained both positive and negative knowledge, but our implementation is a simplified version maintaining only positive information [23].

The issue of let expressions in supercompilation has not previously been a primary focus. If lets are mentioned, the usual strategy is to substitute all linear lets and residuate all others. Lets have been considered in a strict setting [8], where they are used to preserve termination semantics, but in this work all strict lets are inlined without regard to loss of sharing. Movement of lets can have a dramatic impact on performance: carefully designed let-shifting transformations give an average speedup of 15% in GHC [20], suggesting let expressions are critical to the performance of real programs.

Partial evaluation. There has been a lot of work on partial evaluation [7], where a program is specialised with respect to some static data. The emphasis is on determining which variables can be entirely computed at compile time, and which must remain in the residual program. Partial evaluation is particularly appropriate for specialising an interpreter with an expression tree to generate a compiler automatically, often with an order of magnitude speedup, known as the First Futamura Projection [4]. Partial evaluation is not usually able to remove intermediate data structures. Our method is certainly less appropriate for specialising an interpreter, but in the absence of static data, is still able to show improvements.

Deforestation. The deforestation technique [31] removes intermediate lists in computations. This technique has been extended in many ways to encompass higher order deforestation [10] and work on other data types [3]. Probably the most practically motivated work has come from those attempting to restrict deforestation, in particular shortcut deforestation [5], and newer approaches such as stream fusion [2]. In this work certain named functions are automatically fused together. By rewriting library functions in terms of these special functions, fusion occurs.

Whole Program Compilation. The GRIN approach [1] uses whole program compilation for Haskell. It is currently being implemented in the jhc compiler [12], with promising initial results. GRIN works by first removing all functional values, turning them into case expressions, allowing subsequent optimisations. The intermediate language for jhc is at a much lower level than our Core language, so jhc is able to perform detailed optimisations that we are unable to express.

Lower Level Optimisations. Our optimisation works at the Core level, but even once efficient Core has been generated there is still some work before efficient machine code can be produced. Key optimisations include strictness analysis and unboxing [19]. In GHC both of these optimisations are done at the Core level, using a Core language extended with unboxed types. After this lower level Core has been

generated, it is then compiled to STG machine instructions [15], from which assembly code is generated. There is still work being done to modify the lowest levels to take advantage of the current generation of microprocessors [11]. We rely on GHC to perform all these optimisations after Supero generates a residual program.

7 Conclusions and Future Work

Our supercompiler is simple – the Core transformation is expressed in just 300 lines of Haskell. Yet it replicates many of the performance enhancements of GHC in a more general way. We have modified some of the techniques from supercompilation, particularly with respect to let bindings and generalisation. Our initial results are promising, but incomplete. Using our supercompiler in conjunction with GHC we obtain an average runtime improvement of 16% for the imaginary section of the nofib suite. To quote Simon Peyton Jones, "an average runtime improvement of 10%, against the baseline of an already well-optimised compiler, is an excellent result" [18].

There are three main areas for future work:

More Benchmarks. The fifteen benchmarks presented in this paper are not enough. We would like to obtain results for larger programs, including all the remaining benchmarks in the nofib suite.

Runtime Performance. Earlier versions of Supero [13] managed to obtain substantial speed ups on benchmarks such as exp3_8. The Bernouilli benchmark is currently problematic. There is still scope for improvement.

Compilation Speed. The compilation times are tolerable for benchmarking and a final optimised release, but not for general use. Basic profiling shows that over 90% of supercompilation time is spent testing for a homeomorphic embedding, which is currently done in a naïve manner – dramatic speedups should be possible.

The Programming Language Shootout[2] has shown that low-level Haskell can compete with low-level imperative languages such as C. Our goal is that Haskell programs can be written in a high-level declarative style, yet still perform competitively.

Acknowledgements. We would like to thank Simon Peyton Jones, Simon Marlow and Tim Chevalier for help understanding the low-level details of GHC, and Peter Jonsson for helpful discussions and presentation suggestions.

References

1. Boquist, U., Johnsson, T.: The GRIN project: A highly optimising back end for lazy functional languages. In: Kluge, W. (ed.) IFL 1996. LNCS, vol. 1268, pp. 58–84. Springer, Heidelberg (1997)
2. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: From lists to streams to nothing at all. In: Proc ICFP 2007, October 2007, pp. 315–326. ACM Press, New York (2007)

[2] `http://shootout.alioth.debian.org/`

3. Coutts, D., Stewart, D., Leshchinskiy, R.: Rewriting Haskell strings. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 50–64. Springer, Heidelberg (2006)
4. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Higher-Order and Symbolic Computation 12(4), 381–391 (1999)
5. Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: Proc FPCA 1993, June 1993, pp. 223–232. ACM Press, New York (1993)
6. Golubovsky, D., Mitchell, N., Naylor, M.: Yhc.Core - from Haskell to Core. The Monad.Reader (7), 45–61 (2007)
7. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. In: Partial Evaluation and Automatic Program Generation. Prentice-Hall International, Englewood Cliffs (1993)
8. Jonsson, P.A., Nordlander, J.: Positive Supercompilation for a higher order call-by-value language. In: Proc. IFL 2007 (September 2007)
9. Leuschel, M.: Homeomorphic embedding for online termination of symbolic methods. In: The essence of computation: complexity, analysis, transformation, pp. 379–403. Springer, Heidelberg (2002)
10. Marlow, S.: Deforestation for Higher-Order Functional Programs. PhD thesis, University of Glasgow (1996)
11. Marlow, S., Yakushev, A.R., Peyton Jones, S.: Faster laziness using dynamic pointer tagging. In: Proc. ICFP 2007, October 2007, pp. 277–288. ACM Press, New York (2007)
12. Meacham, J.: jhc: John's haskell compiler (2007), `http://repetae.net/john/computer/jhc/`
13. Mitchell, N., Runciman, C.: Supero: Making Haskell faster. In: Proc. IFL 2007 (September 2007)
14. Partain, W., et al.: The nofib Benchmark Suite of Haskell Programs (2007), `http://darcs.haskell.org/nofib/`
15. Peyton Jones, S.: Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. JFP 2(2), 127–202 (1992)
16. Peyton Jones, S.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Engineering theories of software construction, Marktoberdorf Summer School (2002)
17. Peyton Jones, S.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
18. Peyton Jones, S.: Call-pattern specialisation for Haskell programs. In: Proc. ICFP 2007, October 2007, pp. 327–337. ACM Press, New York (2007)
19. Peyton Jones, S., Launchbury, J.: Unboxed values as first class citizens in a non-strict functional language. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 636–666. Springer, Heidelberg (1991)
20. Peyton Jones, S., Partain, W., Santos, A.: Let-floating: Moving bindings to give faster programs. In: Proc. ICFP 1996, pp. 1–12. ACM Press, New York (1996)
21. Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the rules: Rewriting as a practical optimisation technique in GHC. In: Proc. Haskell 2001, pp. 203–233. ACM Press, New York (2001)
22. Röjemo, N.: Highlights from nhc - a space-efficient Haskell compiler. In: Proc. FPCA 1995, pp. 282–292. ACM Press, New York (1995)
23. Secher, J.P., Sørensen, M.H.B.: On perfect supercompilation. In: Bjørner, D., Broy, M., Zamulin, A. (eds.) PSI 1999. LNCS, vol. 1755, pp. 113–127. Springer, Heidelberg (2000)

24. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In: Lloyd, J.W. (ed.) Logic Programming: Proceedings of the 1995 International Symposium, pp. 465–479. MIT Press, Cambridge (1995)
25. The GHC Team. The GHC compiler, version 6.8 (November 2007), `http://www.haskell.org/ghc/`
26. Tolmach, A.: An External Representation for the GHC Core Language (September 2001), `http://www.haskell.org/ghc/docs/papers/core.ps.gz`
27. Turchin, V.F.: The algorithm of generalization in the supercompiler. In: Partial Evaluation and Mixed Copmutation, pp. 341–353. North-Holland, Amsterdam (1988)
28. Turchin, V.F.: Refal-5, Programming Guide & Reference Manual. New England Publishing Co., Holyoke (1989)
29. Turchin, V.F.: The concept of a supercompiler. ACM Trans. Program. Lang. Syst. 8(3), 292–325 (1986)
30. Turchin, V.F., Nirenberg, R.M., Turchin, D.V.: Experiments with a supercompiler. In: Proc. LFP 1982, pp. 47–55. ACM, New York (1982)
31. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: Ganzinger, H. (ed.) ESOP 1988. LNCS, vol. 300, pp. 344–358. Springer, Heidelberg (1988)
32. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc. POPL 1989, pp. 60–76. ACM Press, New York (1989)

Checking Dependent Types Using Compiled Code

Preliminary Report

Dirk Kleeblatt

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
klee@cs.tu-berlin.de

Abstract. Type checkers for dependent types need to evaluate user defined functions during type checking. For this, current implementations typically use an interpreter, which has drawbacks. We show, how at this stage compiled code can be used for a language with lazy evaluation.

1 Introduction

This article gives an early report on the implementation of Ulysses, a lazy functional language with dependent types. These allow a more detailed specification of functional behavior than possible in languages without dependent types. But the increased expressiveness comes at a cost of increased implementation effort.

Ulysses is quite similar to Cayenne [1]. One of the similarities is, that there is no sharp distinction between terms and types, the only difference is that some expressions may be used as types while others may not. The consequences of identifying terms and types are significant: functions can be used not only to construct the usual terms like natural numbers, lists and so on, but also to construct types. Hence, we need to evaluate some user defined functions during type checking time. To circumvent the drawbacks of interpreted code, we use compilation to native machine code instead, which is complicated by the special requirements of evaluation for dependent type checking, namely evaluation under λ abstractions and case analyses. To our knowledge, this is the first implementation using compiled code during type checking for a lazy language. Earlier work exists that is restricted to eager evaluation [2].

In the following, we introduce the language Ulysses (Sect. 2). We adapt former work on strict languages to the needs of lazy evaluation (Sect. 3), and describe the necessary compilation technique (Sect. 4) and runtime system (Sect. 5). This leads to a working system, but we implemented some further improvements (Sect. 6). We give some notes on implementation details (Sect. 7), a comparison with related work (Sect. 8) and directions for future research (Sect. 9).

2 A Description of Ulysses

It is common in functional languages to have type constructors which may be regarded as functions from types to types. In Haskell, for examples, `Maybe` can

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 165–182, 2008.

be applied to the type `Integer`, yielding a new type `Maybe Integer`. But the possibilities to define type constructors are usually restricted to the definition of parametric algebraic data types. In Ulysses, such restrictions do not exist. It is possible to write arbitrary functions that take terms to new types. Comparable with Cayenne, type checking Ulysses programs is not decidable. The user has to ensure, that no infinite recursions are used in functions at the type level. We believe that this not a defect of our design, since we expect that in the same manner that programmers nowadays have to ensure not to write nonterminating programs, they will be able learn how to implement terminating type level functions. As a debugging aid, it is possible to add a timeout to the type checker, and to indicate the type expression that caused the lengthy evaluation. The programmer would have to search for a reason for a nontermination or increase the amount of time he is willing to wait.

An example for recursive type definitions in Ulysses is shown in Fig. 1. The syntax is similar to Haskell. Line 2 gives the well known definition of Peano numbers. Line 1 gives a type declaration for `nat`, `#0` is the type of all (small) types. More interesting is the definition of `vector` in Lines 4–6. The first argument to `vector` is of type `#0`, so it is a type by itself. The second argument is a natural number, and the result type is `#0` again, so this function computes a type when given a term of type `nat` as an argument. This computed type is defined by recursion, the base case is shown in Line 5: A vector of length zero can only be created by the constructor `Nil` which takes no arguments. Line 6 defines a vector of length `S x` to be created by the constructor `Cons`, which prepends an

```
1  nat :: #0;
2  nat = data Z | S nat;
3
4  vector :: #0 -> nat -> #0;
5  vector a Z     = data Nil;
6  vector a (S x) = data Cons a (vector a x);
7
8  add :: nat -> nat -> nat;
9  add Z     b = b;
10 add (S a) b = S (add a b);
11
12 append :: forall a :: #0 .
13    forall n :: nat . vector a n ->
14    forall m :: nat . vector a m ->
15    vector a (add n m);
16 append a Z     Nil          m vm = vm;
17 append a (S p) (Cons ft rt) m vm =
18    Cons ft (append a p rt m vm)
```

Fig. 1. Vectors as lists with fixed length

element of type `a` (a parameter to our definition) to a vector of length `x`. So a value of type `vector a n` is guaranteed to contain exactly `n` elements of type `a`.

Lines 8–10 define the addition of natural numbers as usual. Lines 12–18 give the definition of the append function for vectors. Using dependent types, its type declaration gives a good specification of this function. The type can be read as follows: for any type `a`, given a natural number `n`, and a vector containing `n` elements of type `a`, and furthermore a second natural number `m` together with a vector of size `m`, return a vector containing `n` plus `m` elements.

In the definition of the recursive case in Lines 17 and 18, the resulting type `vector a (add n m)` can be narrowed using the information from the pattern matching, `n` must be equal to `S p`, so we can do the following reductions which are necessary to type check the right hand side:

```
vector a (add (S p) m)
↦ vector a (S (add p m))
↦ data Cons a (vector a (add p m))
```

To perform these reductions, implementations typically use an interpreter during type checking. But this has several disadvantages:

- Interpreted code has reduced performance compared to compiled code.
- When writing a compiler, an additional interpreter is needed just for type checking, and when the language is extended later on, two different parts of code have to be adapted: the compiler as well as the interpreter.
- This gets worse in the presence of even small differences in the semantics of the interpreter and the compiler: computations giving a different result at runtime than during type checking will most probably violate type safety.

However, when replacing an interpreter by a compiler in this setting, we have to keep in mind that the terms that have to be reduced may contain free variables: in the above reductions, all redexes contain not only the type variable `a`, but also two unknown numbers, `p` and `m`. Code generated by usual compilers cannot handle these free variables. Moreover, it is necessary to compute strong normal forms, while code generated by usual compilers computes only weak head normal forms. Hence, a compiler for our type checker for dependently typed languages must cope with evaluation under λ abstraction and case analyses.

Specification of Ulysses. The abstract grammar of Ulysses is shown in the upper part of Fig. 2. The syntactic category e of expressions contains (in this order) variables, constructors, applictions, abstractions, type universes, function types, dependent products and data types. An Ulysses program consists of a set of declarations (*decl*), each consisting of a type signature $x :: e$ and a set of equations defining x (eq^x) via pattern matching over several patterns (p).

The lower part of Fig. 2 shows the evaluation rules of Ulysses. Substituting e for x in e' is denoted by $e'[x/e]$. Besides β reductions, equations can be unfolded when a applicable equation is contained in the program $\mathcal{P}$ under consideration, when the function arguments match the corresponding patterns. The matching relation $\triangleleft$ (not defined here) reduces subexpression if this is necessary

$$
\begin{aligned}
e &::= x \mid C \mid e_1 e_2 \mid \lambda x.e && (1)\\
&\mid \texttt{\#}n \mid e_1 \texttt{ -> } e_2 \mid \texttt{forall } x :: e_1.e_2 && (2)\\
&\mid \texttt{data } (C_1\ e_{1,1} \ldots e_{1,l}) \ldots (C_m\ e_{m,1} \ldots e_{m,k}) && (3)\\
\mathit{decl} &::= x :: e \quad eq_1^x \ldots eq_n^x && (4)\\
eq^x &::= x\ p_1 \ldots p_n = e && (5)\\
p &::= C\ p_1 \ldots p_n \mid x && (6)
\end{aligned}
$$

$$
\textsc{Beta}\ \frac{}{(\lambda x.e_1)e_2 \mapsto e_1[x/e_2]}
$$

$$
\textsc{Equation}\ \frac{(x\ p_1 \ldots p_n = b) \in \mathcal{P} \qquad p_1 \ldots p_n \lhd_\sigma e_1 \ldots e_n}{x\ e_1 \ldots e_n \mapsto \sigma(b)}
$$

$$
\textsc{Context}\ \frac{e \mapsto e'}{K[e] \mapsto K[e']}
$$

$$
\begin{aligned}
K &::= [] \mid C\ e_1 \ldots e_n\ [] \mid Ke \mid \lambda x.K \mid e \texttt{ -> } K \mid K \texttt{-> } e && (7)\\
&\mid \texttt{forall } x :: K.e \mid \texttt{forall } x :: e.K && (8)\\
&\mid \texttt{data } (C_1\ e_{1,1} \ldots e_{1,l}) \ldots K \ldots (C_m\ e_{m,1} \ldots e_{m,k}) && (9)
\end{aligned}
$$

Fig. 2. Abstract grammar and evaluation of Ulysses programs

$$
\textsc{Conv}\ \frac{\Gamma \vdash e :: t_1 \qquad t_1 \leftrightarrow^* t_2}{\Gamma \vdash e :: t_2}
$$

$$
\textsc{Var}\ \frac{x :: t \in \Gamma}{\Gamma \vdash x :: t}
$$

$$
\textsc{Cons}\ \frac{(C\ t_1 \ldots t_n) \quad \in \quad \{(C_1\ e_{1,1} \ldots e_{1,l}), \ldots, (C_m\ e_{m,1} \ldots e_{m,k})\}}{\Gamma \vdash C :: t_1 \texttt{ -> } \ldots \texttt{ -> } t_n \texttt{ -> data } (C_1\ e_{1,1} \ldots e_{1,l}) \ldots (C_m\ e_{m,1} \ldots e_{m,k})}
$$

$$
\textsc{App}\ \frac{\Gamma \vdash e_1 :: \texttt{forall } x :: t_1.t_2 \qquad \Gamma \vdash e_2 :: t_1}{\Gamma \vdash e_1 e_2 :: t_2[x/e_2]}
$$

$$
\textsc{Abs}\ \frac{\Gamma, x_1 :: t_1 \vdash e :: t_2[x_2/x_1]}{\Gamma \vdash \lambda x_1.e :: \texttt{forall } x_2 :: t_1.t_2}
$$

$$
\textsc{Univ}\ \frac{}{\Gamma \vdash \texttt{\#}n :: \texttt{\#}n + 1}
$$

$$
\textsc{Forall}\ \frac{\Gamma \vdash t_1 :: \texttt{\#}n_1 \qquad \Gamma, x :: t_1 \vdash t_2 :: \texttt{\#}n_2 \qquad m = \max(n_1, n_2)}{\Gamma \vdash \texttt{forall } x :: t_1.t_2 :: \texttt{\#}m}
$$

$$
\textsc{Data}\ \frac{\Gamma \vdash t_{1,1} :: \texttt{\#}n_{1,1} \quad \ldots \quad \Gamma \vdash t_{m,k} :: \texttt{\#}n_{m,k} \qquad m = \max(n_{1,1}, \ldots, n_{m,k})}{\Gamma \vdash \texttt{data } (C_1\ t_{1,1} \ldots t_{1,l}) \ldots (C_m\ e_{m,1} \ldots t_{m,k}) :: \texttt{\#}m}
$$

Fig. 3. Typing rules for Ulysses expressions

to decide whether a pattern matches an expression, and yields a substitution σ, which is applied to the right hand side of the defining equation. Reduction takes place in all evaluation contexts K. While $\mapsto$defines call by name reduction, our implementation uses subexpression sharing to implement call by need.

The rules specifying the type system are shown in Fig. 3. For better readability, we use two different meta-variables e and t both ranging over expressions, choosing t when the expression is used as a type. The function space is a spacial case of the dependent product constructed by `forall` when the bound variable is not used in the result type, and hence not shown in the typing rules.

Rule Conv, using the transitive, reflexive and symmetric closure $\leftrightarrow^*$of the reduction relation $\mapsto$, is the key rule for our work, enforcing evaluation during type checking.

3 Weak Normalization and Readback

Our approach does not compute strong normal forms of expressions in one big step. Instead, we use an adopted version of a well known weak head evaluator, examine the weak head normal forms computed by this evaluator, and extract remaining redexes. These can then be recursively reduced.

This proceeding is not new, it was introduced by Grégoire and Leroy [2]. However, they restricted their focus on strict evaluation, using the ZAM abstract machine as a weak evaluator. Our work employs lazy evaluation, taking the spineless tagless g-machine by Peyton Jones [3] as a weak evaluator. The differences between ZAM and STG machine are quite significant, so transferring the existing work from strict evaluation to lazy evaluation is nontrivial.

We adopt the definition from [2] of the strong normalization function $\mathcal{N}$ to the needs of the STG machine. This function is defined in terms of two helper functions. The weak evaluation function $\mathcal{V}$ reduces terms to weak head normal forms, the readback function $\mathcal{R}$ scrutinizes the resulting weak head normal form, extracts remaining redexes, and applies $\mathcal{N}$ recursively on unevaluated subterms.

Definition 1. *The normalization function $\mathcal{N}$ is defined as*

$$\mathcal{N}(e) = \mathcal{R}(\mathcal{V}(e)).$$

The weak evaluation of Ulysses expressions is done by compiling them first to STG code and then to native machine code (cf. Sect. 4). This machine code is executed, and the execution stops when a weak head normal form is reached. Its structure can then be extracted by interpreting the final machine state (cf. Sect. 5).

In the following definition, the subexpressions e and e_i are *machine representations* of expressions, i.e. closure pointers into the heap.

Definition 2. *A weak head normal form is given by one of four cases:*

$$v ::= C\ e_1\ \ldots\ e_n \tag{10}$$

$$\mid \; x\ e_1\ \ldots\ e_n \tag{11}$$

$$\mid \; \lambda x\,.\,e \tag{12}$$

$$\mid \; \texttt{case}\ \Sigma[\,x\ e_1\ \ldots\ e_n\,] \tag{13}$$

Line 10 describes the application of a constructor to n argument expressions ($n = 0$ for nullary constructors), while Line 11 denotes a free variable x, again applied to n arguments ($n = 0$ for no arguments). Line 12 is a unsaturated function, i. e. a function expecting at least one additional argument.

The last case, number 13, is encountered when a case distinction is to be made to implement pattern matching, and the scrutinee is not a constructor term, but an application of a free variable to zero or more arguments. The context Σ fixes the continuation for each possible constructor, i. e. the resulting computation after the case distinction. It is defined as the state of the machine stacks and registers, according to Sect. 4.3.

Next, we define the readback function $\mathcal{R}$ by case analysis on weak head normal forms.

Definition 3. *The readback function $\mathcal{R}$ is defined as*

$$\mathcal{R}(C\ e_1\ \ldots\ e_n) = C\ \mathcal{N}(e_1)\ \ldots\ \mathcal{N}(e_n) \tag{14}$$

$$\mathcal{R}(x\ e_1\ \ldots\ e_n) = x\ \mathcal{N}(e_1)\ \ldots\ \mathcal{N}(e_n) \tag{15}$$

$$\mathcal{R}(\lambda x\,.\,e) = \lambda y\,.\,\mathcal{N}((\lambda x\,.\,e)\ y) \quad (y\ \textit{fresh}) \tag{16}$$

$$\mathcal{R}(\texttt{case}\ \Sigma[\,x\ e_1\ \ldots\ e_n\,]) = \texttt{case}\ x\ \mathcal{N}(e_1)\ \ldots\ \mathcal{N}(e_n)\ \texttt{of} \tag{17}$$
$$\{\ C_i\ \vec{x}_i \to \mathcal{N}(\Sigma[\,C_i\ \vec{x}_i\,])\ \}$$
(where C_i are the possible constructors and $\vec{x}_i$ fresh)

Equation 14 neatly shows the idea of strong normalization by weak evaluation and readback: When a constructor term has been evaluated, the resulting normal form again is a constructor term. However, the constructor arguments $e_1\ \ldots\ e_n$ have not necessarily been evaluated in the first step, because we employ lazy evaluation. Hence, these arguments have to be normalized by $\mathcal{N}$, which results in their (weak) evaluation and subsequent readback (cf. the definition of $\mathcal{N}$ in Def. 1), which in turn might result in further weak evaluations, and so on.

Equation 15 is completely analogous: when a free variable is applied to zero or more arguments, we have to normalize these arguments.

In equation 16, evaluation under λ is described. We generate a fresh variable y, normalize the expression $(\lambda x\,.\,e)\ y$ where y is free, and the resulting normal form is placed beneath a λ abstraction. This might look complicated at a first glance, an usual definition would involve substitution, resulting in the normalization of $e[x/y]$ instead of the application $(\lambda x\,.\,e)\ y$. However, it is important that the abstraction $\lambda x\,.\,e$ given as argument to $\mathcal{R}$ remains unchanged on the right hand side of the definition. Recall that this abstraction is represented by heap closures of our weak evaluator. It is quite comfortable not being forced to define substitution on these structures, while it is quite easy to push additional arguments on a machine stack.

When evaluation gets stuck because a case analysis of an application of a free variable cannot be reduced further, we have to evaluate all possible branches of the analysis, as shown in Def. 17. We generate all possible constructors C_i, applied to fresh variables $\vec{x}_i$ according to their arity. Which constructors (and with which arity) are needed can be deduced from the type of the free variable x and its arguments. Details to this can be found in Sect. 5.

Every new constructor application is then executed in context Σ, where executing an expression e in context Σ, i.e. $\Sigma[e]$, means entering the closure for e in the machine state given by Σ (cf. Sect. 5). As mentioned, this context captures the branches for the case analysis, so each $\Sigma[\, C_i\ \vec{x}_i\,]$ is normalized to the corresponding case arm for $C_i\ \vec{x}_i$. Since we use lazy evaluation, the arguments $e_1\ \ldots\ e_n$ may be unevaluated, so we normalize them using $\mathcal{N}$, too.

4 The Spineless Tagless G-Machine

Up to now, we treated the weak evaluation function $\mathcal{V}$ fairly abstract. In this section we describe the spineless tagless g-machine, which we used to implement a lazy weak evaluator. A more complete description can be found in [3]. Here, we will focus on the aspects of the STG machine that are relevant for our modifications and the machine state interpretation we detail in Sect. 5.

Every Ulysses expression that has to be normalized is translated to native machine code in two steps: first, we create STG code, which looks like a restricted and annotated functional programming language, and from this we generate target machine assembly code.

4.1 The STG Language

STG code is formed according to the grammar in Fig. 4. We describe a simplified variant of STG code. The original formulation is prepared for *primitive values* to deal e.g. with unboxed integers. Additionally, we do not distinguish recursive and nonrecursive `let` bindings, we treat all bindings as recursive.

The first form of STG expressions is a constructor application (Line 18 in Fig. 4). However, the constructor has to be *saturated*, i.e. all arguments according to the arity of the constructor C have to be present. Moreover, the constructor arguments are restricted to be variables.

A function application (Line 19) is restricted in a similar manner: the function x and all arguments given have to be variables. No anonymous functions exist in this intermediate language, they have to be bound globally or locally. Function applications do not have to be saturated but can be partial.

Local definitions are bound by `let` expressions (Line 20). Each binding b associates a so-called *lambda form* of the syntactic category lf with a name. A lambda form (Line 23) is annotated with two lists of variables. It abstracts over the variables $y_1\ \ldots\ y_m$, defining an m-ary function. The list $x_1\ \ldots\ x_n$ gives the free variables of the body e, excluding the abstracted variables y_i.

Additionally, each lambda form is annotated with an *update flag* π which can be `u` or `n`. These flags are necessary for the implementation of *lazy evaluation*,

$$
\begin{aligned}
e &::= C\ \{x_1\ \ldots\ x_n\} && (18)\\
&\mid\ x\ \{x_1\ \ldots\ x_n\} && (19)\\
&\mid\ \texttt{let}\ bs\ \texttt{in}\ e && (20)\\
&\mid\ \texttt{case}\ e\ \texttt{of}\ as && (21)\\
bs &::= x_1 = lf_1;\ \ldots\ ; x_n = lf_n && (22)\\
lf &::= \{x_1\ \ldots\ x_n\}\ \backslash\pi\ \{y_1\ \ldots\ y_m\}\ \rightarrow\ e && (23)\\
\pi &::= \texttt{u} \mid \texttt{n} && (24)\\
as &::= a_1;\ \ldots\ ; a_n;\ d && (25)\\
a &::= C\ \{x_1\ \ldots\ x_n\}\ \rightarrow\ e && (26)\\
d &::= \_\ \rightarrow\ e && (27)
\end{aligned}
$$

Fig. 4. Grammar for STG code

where each closure is evaluated only when necessary, but at most once. To ensure this, closures are overwritten with their weak head normal form after their first evaluation. However, not every closure has to be overwritten: if the bound expression already is in weak head normal form (i.e. an abstraction with nonempty $\{y_1\ \ldots\ y_m\}$, or a constructor application), or the compiler can prove that it will be evaluated only once anyway, the binding is flagged with `n` to signal that no update code has to be generated.[1] Otherwise, the binding is flagged with `u` to cause the generation of update code. For example, in the expression

```
let compose = {} \n {f g x} →
      let gx = {g x} \u {} → g {x}
      in f {gx}
in ...
```

`compose` is defined in weak head normal form, since it abstracts over `f`, `g` and `x`, and is flagged with `n` accordingly. By contrast `gx` is *not* in weak head normal form, `g` and `x` are merely free variables, so the flag is `u` and the closure of `gx` will be overwritten as soon as it is evaluated the first time[2].

Case analysis (Line 21) can be done on arbitrary expressions, but is restricted to flat patterns without nesting. The default case is expressed using the pattern `_`, matching every expression.

4.2 Translating Ulysses to STG Code

The translation of Ulysses code to the STG language is straight forward. Function arguments that are not yet simple variables are bound by new local variables. The same holds for constructor arguments, furthermore we have to saturate

[1] Avoiding unnecessary updates enhances efficiency.
[2] Of course, each application of `compose` will create a new closure `gx`.

constructors by η expansion: a binary constructor `Pair` applied to a single argument `x`, i.e. `Pair x`, becomes

```
let f = {x} \n {y} → Pair x y in f.
```

Nested patterns from Ulysses definitions are flattened by a well known pattern matching compiler, as described in [4].

Besides the usual feature set at the term level, we need a representation of Ulysses types. This is done by introducing a reserved constructor for each predefined type constructor. The simplest case is the function space, a type `a -> b` is translated to the constructor application `Fun a b`.

The encoding of type universes and data types makes use of unboxed integers. For instance, we represent `#2` by `Universe 2`, and e.g. `data Nothing | Just a` as `Data 1 2 a`. In the latter case, 1 and 2 are the tags for the constructors `Nothing` and `Just`, and their arity can be seen by the number of boxed values after the constructor tag, in this case 0 and 1, respectively. Note that this is the only place where we make use of unboxed integers, as we implemented only limited support for them in our prototype implementation.

Dependent product types are represented by the reserved constructor `Forall`, taking as arguments the representations of argument and result types. To make the necessary substitutions in the result type possible, we use a technique that was used already in [5]. The result type is not stored directly, but as a function taking a member of the argument type to a type representation. The Ulysses type of the identity function `forall t :: #0 . t -> t` is thus represented as

```
let x = {} \n {} → Universe 0;
    y = {} \n {t} → Fun t t
in Forall x y
```

which allows to replace `t` during type checking with a concrete type by extracting `y` from the constructor expression, and applying it to the needed type.

4.3 Executing STG Code on Conventional Machines

STG code can be easily translated to machine code for execution on traditional hardware. We next give an overview of the memory layout and operational behavior of the resulting machine programs.

Our machine state Σ consists of

- a *heap* which contains closures, each consisting of one code pointer, a sequence of pointers to the values of the free variables used in this code, and two words with meta information about the closure, namely the size, and whether a variable is bound to an unboxed integer instead of a pointer (only used for the implementation of the reserved constructor `Data`, cf. Sect. 4.2),
- a *closure register* $\mathtt{R}_{\mathtt{closure}}$, pointing to the currently evaluated closure,
- an *argument stack*, containing pointers to closures in the heap, for passing arguments to functions,

- a *continuation stack*, holding code pointers, and a *tag return register* $\texttt{R}_{\texttt{tag}}$, containing a small integer assigned to the individual constructors for the implementation of case analyses, and
- an *update stack* of update frames, for bookkeeping closures that have to be overwritten as soon as they are evaluated to weak head normal form.

Function Application. We implement function calls following the push/enter model: we push all arguments onto the argument stack, and enter the function. Entering a function is done in two steps: first, load the address of the function's closure into the closure register, and second jump to the function body. Since the STG language does not allow nested function applications, this is a tail call, and no return address has to be remembered. So we translate an STG function application `f {x y}` to following pseudo assembler code:

```
push-argument y
push-argument x
enter f
```

Constructors and Case Analyses. Constructor applications usually occur as scrutinees within case analyses. When a `case` expression is evaluated, a return address is pushed onto the continuation stack. Next, the evaluation of the scrutinee is started. When the scrutinee is finally evaluated to a constructor application, the constructor tag is loaded into the tag return register $\texttt{R}_{\texttt{tag}}$ and a pointer to a closure containing the constructor arguments is loaded into the closure register $\texttt{R}_{\texttt{closure}}$. These registers now have to be passed to the code of the case analysis, so a jump to the topmost code pointer on the continuation stack is taken. At the jump target, the continuation is removed from the stack, and the tag is analyzed. The constructor arguments can be accessed through the closure register. Accordingly, the STG expression

$$\texttt{case } e_1 \texttt{ of \{ C x y} \rightarrow e_2\texttt{; \_} \rightarrow e_3 \texttt{ \}}$$

is compiled to the following pseudo assembler:

```
   push-continuation l
   ≪ code for e1 ≫
l: pop-continuation
   compare Rtag ≪ tag reserved for C ≫
   jump-if-not-equal d
   ≪ code for e2 ≫
d: ≪ code for e3 ≫
```

and a corresponding constructor application `C {a b}` is translated to[3]

```
   Rclosure := allocate l, {a b}
l: Rtag := ≪ tag reserved for C ≫
   jump-continuation
```

[3] Here, `allocate` allocates heap space for a new closure and fills it with the given code pointer and free variables.

Local Bindings. For `let` bindings, we allocate on the heap a closure for each bound variable. The code pointers of these closures point to the compiled bodies of the lambda forms. The current values of the free variables, which are pointers to other closures, are saved into the corresponding closure fields. After that, evaluation continues with the body of the `let` expression. Note that due to the lazy semantics no evaluation of the bound variables is triggered now. A binding with update flag `n` as e. g.

`let v = {x y} \n {}` $\rightarrow e_1$ `in` e_2

is thus translated to

```
    allocate l, {x y}
    ≪ code for e2 ≫
l: ≪ code for e1 ≫
```

When the closure shall be updated after its first evaluation, the code of the new closure is preceeded by pushing an update frame that contains the current closure pointer (pointing to the memory location to be overwritten), and the current argument stack content[4]. Next, the argument stack is emptied to signal a necessary update to partial function applications. Thus a binding with update flag `u` as e. g.

`let v = {x y} \u {}` $\rightarrow e_1$ `in` e_2

is compiled to

```
    allocate l, {x y}
    ≪ code for e2 ≫
l: push-update-frame
    empty-argument-stack
    ≪ code for e1 ≫
```

Accordingly, each function checks whether all expected arguments are present and, if not, calls a global routine `updatePAP` (*update partial application*) that

- overwrites the closure pointed to by the topmost update frame with a partially applied function closure,
- removes the update frame from the update stack,
- restores the argument stack,
- and finally re-enters the current closure.

So we translate

`let v = {} \n {x y}` $\rightarrow e_1$ `in` e_2

to the assembly code

```
    allocate l, {x y}
    ≪ code for e2 ≫
l: compare-argument-stack-length 2
    jump-if-less updatePAP
    ≪ code for e1 ≫
```

[4] In [3] you can find a description how this can be done by fast pointer manipulations.

This allows to update closures with function values, but we need to find a way for constructor values, too. This can be done quite elegant by merging the update and the continuation stack, pushing update frames and continuation onto the same stack. Now, we can arrange update frames so that the topmost word on the stack contains a pointer to a routine `updateConstructor` that overwrites the closure pointed to by the update frame with an indirection to the current closure, removes the update frame, and jumps to the now topmost pointer on the stack. So a constructor can simply jump to the topmost pointer on the merged stack, which points either to the update routine or to the case analysis code.

5 Runtime System

Evaluating type expressions to normal forms by running machine code requires a special runtime system with two main tasks:

- The different weak head normal forms (cf. Def. 2) have to be discriminated, and their components have to be extracted from the machine state.
- To evaluate under λ and `case`, we need to generate free variables. They are carefully designed to fit to the generated machine code, so that we are not forced to generate special machine code that deviates from traditional STG compilers and might have poor performance.

5.1 Constructor Expressions

To identify final machine states that constitute constructor expressions we exploit that each constructor, after loading the tag return register and closure pointer, just takes a jump to the topmost address on the continuation stack. So, before we start running any machine code, we just push a special exit continuation on the continuation stack. The corresponding code finds the constructor tag and arguments via the respective registers, and can return them to the readback function (cf. Def. 3) for recursive evaluation of the constructor arguments.

5.2 Unsaturated Functions

To recognize unsaturated functions, we utilize the fact that each function starts evaluation by checking whether enough arguments are present on the stack. If this check fails, this usually means that an update has to be performed and the global function update routine is jumped to. When an update frame is found, a closure is overwritten and additional arguments are restored on the stack. But if the update stack is empty, we know that the weak head normal form of the overall expression is an unsaturated function, so we return a λ abstraction to the readback function, which is responsible for creating a fresh free variable, pushing it onto the argument stack and reentering the last evaluated closure.

At this point we have introduced an additional check compared to code generated by traditional compilers using STG intermediate code: when not enough

function arguments are present on the argument stack, we have to check whether the update stack is empty. Usually, a restricted top level type ensures that this does not happen. However, this check occurs only at a single code location in `updatePAP`. So it is possible to link this routine during runtime, opposed to evaluation at type checking time, to a simpler version omitting this check.

5.3 Free Variables

Free variables are the most intricate part of our implementation. They originate from evaluations under λ abstractions, where they were pushed onto the argument stack, or evaluations under `case`, where constructor expressions with free variables in argument positions were created.

Code generated from STG intermediate code has a distinguishing property from many other compilation schemes: constructor closures and function closures are entered in the same way, but execute very different code. While the former take a jump to a code address left on the continuation stack, the latter expect arguments on the stack and start some computation. So we need an implementation of free variables prepared for both scenarios.

But luckily one invariant exists in both cases: entering a free variable means a weak head normal form has been reached. We just have to find out, whether it is a free variable application (Def. 2, Line 11) or a case analysis of a free variable application (Def. 2, Line 13), and collect eventual arguments from the stack.

In our system, we implemented free variables as follows. Each free variable is represented by a closure on the heap. All free variables share a common code pointer into the runtime system. Every variable closure contains an identification number, to recognize several occurrences of the same variable. Moreover, every variable is annotated with its type, to allow us to select the right behavior when it is entered. As a last component, every free variable closure contains a list of arguments to which it is applied. This list is empty when a fresh variable is created but, when a closure is updated with a free variable application, some arguments might have been accumulated. Thus, the implementation of free variables has no additional cost for code not containing free variables, i.e. we need no explicit checks before entering closures.

When a free variable is entered, the runtime system starts a loop that interprets the type annotation of the variable in the current machine state. The following algorithm operates on the entered variable x, the collected arguments $e_1 \ \ldots \ e_n$ found in the variable closure, and the type of x applied to $e_1 \ \ldots \ e_n$, also found in the closure. It returns the resulting weak head normal form, which is a free variable application or a case analysis on a free variable application.

1. if the x applied to $e_1 \ \ldots \ e_n$ has a function type, then
   (a) if there is an argument on the argument stack, then
      i. pop this argument, and add it to the accumulated arguments
      ii. compute the result type of the variable's type
      iii. restart at 1 interpreting this result type
   (b) else, if there is an update frame on the update stack

i. perform an update, overwriting the destination closure with a free variable closure containing all arguments accumulated so far and the type of this application
ii. restore argument and return stack from the update frame
iii. restart at 1

(c) else, the free variable applied to the accumulated arguments constitutes the weak head normal form that is returned

2. else, if the variable has a data type
   (a) if there is a continuation on the continuation stack, then
      i. save the current machine state as context Σ
      ii. return a case analysis of a free variable application in context Σ, i.e. `case` $\Sigma[\, x\ e_1\ \ldots\ e_n\,]$
   (b) as in 1b
   (c) as in 1c
3. otherwise, the free variable application x applied to $e_1\ \ldots\ e_n$ is returned

6 Further Improvements

By now, we have all we need for a fairly efficient type checker for dependent types. Type expressions can be reduced by the strong evaluation function $\mathcal{N}$. It uses a weak evaluator $\mathcal{V}$ that is implemented using a approved compilation scheme via STG to assembly code. The results of the weak evaluation can be extracted from the machine state, and all occurring weak head normal forms can be distinguished. By passing these evaluation result to the readback function $\mathcal{R}$, further redexes can be extracted and again passed to our weak evaluator. However, Ulysses implements two additional improvements.

6.1 Interleaving Type Checking and Evaluation of Types

Instead of reducing all types occurring during type checking in one step to normal forms, it is beneficial to reduce them in the first step to weak head normal form only. This weak head normal form is usually a type constructor applied to not yet normalized type arguments. In this stage, we can check whether the type constructor matches the syntactic construct to check, e. g. a function type and a λ abstraction, and reduce the type arguments only when this check succeeds. When this check fails, e. g. because a λ abstraction shall be checked to have a data type, the redexes in the arguments of the type do not have to be reduced, and some amount of unnecessary work can be avoided.

This idea of reduction to weak head normal form interleaved with type checking and recursive further evaluation has been applied in [6], too.

6.2 Detecting Equivalent Types Early

We do type checking only, not considering type inference. So whenever we check a variable x against a type, x has been declared with some type, and thus can

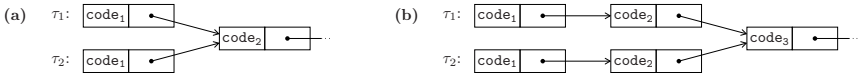


Fig. 5. Detecting equal normal forms without evaluation

be found in the type context Γ. Since we delay reduction of type expressions as long as possible, the type declaration contained in the type context is held only as a weak head normal form. This has to be compared with the type needed by the expression surrounding x, which again is a weak head normal form. If we call the declared type τ_1 and the needed type τ_2, we could reduce τ_1 and τ_2 to normal forms, and compare them for equivalence. But we can do better.

The types τ_1 and τ_2 are represented as closures in the heap. These closures evaluate certainly to the same normal form, if they contain the same pointers in the same places, i. e. the same code pointer and the same pointers for the free variables (cf. Fig. 5 (a)). Moreover, they also evaluate to the same normal form if they have the same code pointer, and their free variable pointers are different, but the respective closures they point to have the same code pointers and the same variable pointers (cf. Fig. 5 (b)).

Generalizing this pattern, we reach a variant of Park's bisimilarity (cf. [7]). The following definition uses the notation c_i to access field i in closure c, $|c|$ for the number of fields of closure c, and $*p$ to dereference a pointer.

Definition 4. *We call a relation R on closures a* bisimulation*, iff for each pair of closures $(s,t) \in R$*

1. *$s_0 = t_0$, i. e. the code pointers are equal, and*
2. *$|s| = |t|$, i. e. the closures have same length, and*
3. *$\forall i \in \{1 \ldots |s|-1\}$:*
   (a) *s_i and t_i are pointers, and $(*s_i, *t_i) \in R$, or*
   (b) *s_i and t_i are non-pointers, and $s_i = t_i$.*

We can consider τ_1 and τ_2 to be equal, if there is a bisimulation R such that $(\tau_1, \tau_2) \in R$. Whether such a relation exists can be checked by a single simultaneous traversal of both closure graphs of τ_1 and τ_2. If this traversal reaches closures with different code pointers, further reductions have to be done.

This notion is appropriate for non-normalizing terms, too. Recursive type definitions as for example `nat = data Z | S nat` can be unfolded infinitely, thus the simple strategy of complete reduction and subsequent equality check is not successful for this type, while the bisimilarity check allows to deal with it.

7 Notes on the Implementation

The Ulysses system has been implemented in Haskell, and is available at `http://uebb.cs.tu-berlin.de/~klee/ulysses`.

7.1 The Overall Type Checking Process

The first step during type checking an Ulysses program is a dependency analysis. It has to ensure that functions, which are used in the types of other function definitions, are type checked before these other definitions. The definitions are sorted accordingly, whereby mutually recursive definitions are clustered together. In each cluster, the type declarations are checked first (since they might contain e.g. wrongly typed function applications, too), before each definition is first checked against the declared type and then compiled to machine code. For compilation, we use Harpy [8], a Haskell library for runtime code generation. It allows to write x86 machine code into memory buffers, and to directly execute it without external tools. Without such a tool, the performance gain of compilation would easily be outweighed by the overhead of e.g. starting a C compiler and loading the resulting object file for each type declaration.

As soon as type checking is completed, the code of the whole checked Ulysses file is located in the Harpy code buffer and could be dumped to an object file. However, this is not yet implemented.

7.2 Copy-on-Write

In Def. 3 Line 17, we had to save the machine state as a context Σ for each case analysis of a free variable. This is no problem for the registers and stacks, as they are fairly small. The heap, however, can be quite large, so unnecessary copies should be avoided when possible. Therefore, we use a copy-on-write approach. Instead of making a copy of the heap, we use the memory management unit of the x86 processors to write protect the heap. When the running program tries to modify the heap, a segmentation violation signal is raised by the operation system. This is handled by our runtime system making a copy and releasing the protection of the affected page, so only modified pages have to be copied.

8 Related Work

Our work is a transfer of the approach of Grégoire and Leroy [2,5] from the strict ZAM abstract machine to the lazy STG machine. Therefore, the readback function had to be adopted. For free variable applications and constructor applications, the eager evaluation strategy resulted in completely evaluated constructor arguments, which can be simply read back by $\mathcal{R}$, while we need a recursive call to the weak evaluator.

The implementation of free variables is quite different than described in [2]. The main reason for these differences lies in the different underlying abstract machines, mainly the uniform handling of constructor and function closures which are both *entered*, which is quite specific to the STG machine.

In [9] Crégut presents an abstract machine for strong normalization of λ terms. It differs from ours and Grégoire's in the missing distinction of weak evaluation and readback. This might be faster when reducing to normal forms, but precludes the improvements of Sect. 6.

Another related line of research is partial evaluation, most closely probably type-directed partial evaluation introduced by Danvy in [10]. This evaluator generates constructors expressions not only when free variables are scrutinized in case expressions, but whenever functions take arguments of disjoint sum types, introducing additional case analyses. These analyses change the strictness properties of normalized expressions, even non-strict functions are normalized to strict functions. This is avoided by our approach of residuating case expressions only when free variables are analyzed by a case expression stemming from a pattern matching in the original source code. Moreover for the same reason our implementation deals better with recursive data types.

The online variant of type-directed partial evaluation [11] introduces the runtime distinction of static versus dynamic values which is typical for online partial evaluation, and which is avoided by our implementation of free variables.

9 Conclusion and Future Work

Ulysses is a prototype of a language with dependent types, that uses compiled code during type checking where interpreters are used traditionally. Even though it misses some features that constitute a complete programming language, as e. g. garbage collection and a module system, it shows the feasibility of our approach.

First experiments suggest that the performance gain is as expected when switching from an interpreted to a compiled system, but further benchmarks have to be done. The performance can be increased by the possibility of separate compilation: instead of compiling type level function before every run of the type checker, a suitable module system would allow us to precompile a module containing the definitions relevant for the type level of other modules.

Benchmarking is complicated by the fact that it is not at all clear how a reasonable benchmark for a type checker should look like. We believe that the slight modifications to the code generation process compared to the classical STG machine influence performance little.

The bisimilarity check (cf. Sect. 6.2) allows us to deal with infinite recursive types, but not all type equivalences can be detected this way. While it works for many types, such as natural numbers, lists and vectors, it needs a great deal of knowledge of system internals to find the reasons why it does not work in some circumstances, leading to a nonterminating type check. A topic for future research is finding a simple criterion which recursive types can be proven equal by bisimilarity after a finite number of reduction steps. Alternatively, integrating an explicit fixpoint operator, as in [2], should improve the situation.

Several optimization techniques for STG code are known, two of them, dynamic pointer tagging and call pattern specialization, recently improved the performance of the ghc compiler. While we believe that most of such techniques can be used in our settings, a closer look is necessary to find possible interactions with the readback scheme and the interpretation of final machine states, as well as with the implementation of free variables.

To be assured that Ulysses is indeed a type safe language, we plan to formalize our reduction and readback scheme and prove its correctness.

An interesting opportunity for future research is the application of our technique to partial evaluation and to proof assistants relying on dependent types, as e.g. Coq (which was the target of Grégoire's work [5]), both areas being equipped with their own specific demands.

References

1. Augustsson, L.: Cayenne – a language with dependent types. In: ICFP 1998: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming, pp. 239–250. ACM Press, New York (1998)
2. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP 2002: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming, pp. 235–246. ACM Press, New York (2002)
3. Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. Journal of Functional Programming 2(2), 127–202 (1992)
4. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Englewood Cliffs (1987)
5. Grégoire, B.: Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml. Thése de doctorat, spécialité informatique, Université Paris 7, école Polytechnique, France (2003)
6. Coquand, T.: An algorithm for type-checking dependent types. Science of Computer Programming 26(1-3), 167–177 (1996)
7. Park, D.: Concurrency and automata on infinite sequences. In: Proceedings of the 5th GI-Conference on Theoretical Computer Science, pp. 167–183. Springer, Heidelberg (1981)
8. Grabmüller, M., Kleeblatt, D.: Harpy: run-time code generation in Haskell. In: Haskell 2007: Proceedings of the ACM SIGPLAN workshop on Haskell workshop, p. 94. ACM, New York (2007)
9. Crégut, P.: An abstract machine for lambda-terms normalization. In: LFP 1990: Proceedings of the 1990 ACM conference on LISP and functional programming, pp. 333–340. ACM Press, New York (1990)
10. Danvy, O.: Type-directed partial evaluation. In: POPL 1996: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 242–257. ACM Press, New York (1996)
11. Danvy, O.: Online type-directed partial evaluation. In: Fuji International Symposium on Functional and Logic Programming, pp. 271–295 (1998)

Debugging Lazy Functional Programs by Asking the Oracle*

Bernd Braßel and Holger Siegel

Department of Computer Science
University of Kiel, 24098 Kiel, Germany
{bbr,hsi}@informatik.uni-kiel.de

Abstract. The complexity of lazy evaluation forbids classic debugging techniques like a simple step-by-step representation of the buggy program run. Therefore, most sophisticated tools for finding bugs in lazy functional programs try to display the run as if the program's underlying semantics was strict. In order to provide such a strict representation, current approaches gather a lot of information about the executed program.

We utilized a new technique to drastically reduce the amount of gathered data and show how to use the reduced information to implement a debugging tool which supports declarative debugging as well as a strict step-by-step tracer.

1 Introduction

The task of designing tools to find bugs in lazy functional programs is demanding. On one hand, the sophisticated strategy employed by the underlying implementation enables the programmer to write code at a high level of abstraction. On the other hand the same sophisticated strategy makes it very hard to understand how a given program is executed step-by-step. Most successful approaches to debugging solve this problem by collecting enough data to represent the program's execution as if the underlying strategy was strict, which is much easier to understand. Examples for such approaches are declarative debugging, cf. [5,6], observations, cf. [4], and redex trailing, cf. [8]. The most comprehensive tool, HAT [2], supports all three approaches among others.

In order to present information about the program in a simple way, the above approaches collect data during program execution. This is especially true with powerful tools, e.g., the HAT system collects megabytes of data in many cases. In [1], we developed an approach to collect considerably less data and still be able to provide the user with a strict view on the execution of his program. The basic idea is that the critical information to replay a lazy computation as if the underlying semantics was strict is when unneeded redexes are discarded. Therefore, we only count the number of strict steps between such discarding steps. We call the resulting list of numbers an *oracle*. Different debugging tools

* This work has been partially supported by DFG grant Ha 2457/1-2.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 183–200, 2008.

```
module Example where

import Prelude hiding (length)

length []     = 0
length (_:xs) = length xs

exp = length (take 2 (fiblist 0))

fiblist x = fib x : fiblist (x+1)

fib :: Int -> Int
fib _ = error "this will not be evaluated"
```

Fig. 1. Example program

can then be realized as strict monadic versions of the original program correctly consuming the number of steps in an oracle. [1] contains a soundness proof for this technique and this paper presents the implementation of a debugging tool based on the oracle technique.

2 Example Sessions

So far, our debugging tool supports two modes. The first is an implementation of the well known declarative debugging method, described in Section 2.1. The second is a step-by-step tracer allowing us to follow a program's execution as if the underlying semantics was strict, skipping uninteresting sub computations. In addition, the tool gives some support for finding bugs in programs employing I/O. This approach to "virtual I/O" is presented along with the step-by-step mode in Section 2.2.

2.1 Declarative Debugging

Figure 1 shows a small example program containing an intentional error to demonstrate the declarative debugging mode. The function `fiblist` creates a potentially infinite list of delayed calls to function `fib`. Due to laziness, `fib` is never evaluated in our example, so we omit its definition. The infinite list is cut to the first two elements by a call to function `take`, which is defined in the usual way. On top level, function `length` is applied to count the elements of the resulting list. It is to be expected that the program returns the number 2.

```
> :l Example
Example> exp
0
```

We see that running the program reveals the result `0`, which indicates that there must be a bug somewhere. Therefore, we switch on the debug mode and execute the program once again.

```
Example> :set +d
Example> exp
```

In the background, our example program and all the modules it depends on are transformed to new modules such that, e.g., the resulting `OracleExample` now depends on `OraclePrelude`.

```
Example> exp
processing: OracleExample
up-to-date: OraclePrelude
```

As we see, our `OraclePrelude` is still up to date, and is not generated again. The program resulting from this transformation behaves exactly like the original program. The only difference is that – as a side effect – it will produce an "oracle". Before continuing with the debugging session we take a look at this oracle.

Evaluating `exp` in `OracleExample` creates a file in the current directory calles `Example.steps`:

```
$ cat Example.steps
[2,0,1,0,0,23]
```

These numbers compactly represent the laziness information. If every expression in the program was evaluated there would have been only a single number. This number indicates how many steps that evaluation would have taken. The fact that there are six numbers for this example tells us that five expressions have been discarded without evaluation. (Two calls to `fib`, two to `(+)` and one to `fiblist`). For more details about the oracle format, how it is produced and why it can be utilized to correctly execute the traced program strictly, cf. [1].

The user does not see anything of the oracle or the steps file. Directly after the steps file has been produced, the debugging tool proceeds by applying a second transformation on the modules.

```
up-to-date: StrictPrelude.hs
generating ./StrictExample.hs
```

The second transformation produces Haskell modules named `Strict*.hs`. These Haskell modules contain the definitions to execute the original program with an underlying strict semantics. The details of this transformation will be presented in Section 3. Upon completion of the second transformation the actual debugging session starts.

```
 ----    ----    -----
(  _ \  (_  _)  (  _  )   Believe
 ) _ <   _)(_    )(_)(    in
(____/()(____)()(_____)() Oracles
--------type ? for help----------

exp
```

Initially, we only see a call to function `exp` which was the main expression in our example. Pressing `i` turns on *inspect mode*. In inspect mode, the result of every sub computation is directly shown and can be "inspected" by the user, i.e., rated as correct or wrong. Inspect mode therefore corresponds to the declarative debugging method. But as we will see in the next section, the display of results of sub computations can be turned on and off at any time. Of course, there is a help menu available, showing a list of all possible inputs.

After pressing `i`, the debugger evaluates the expression and displays the result.

```
exp ~> 0
```

We expected main to have value `2`, but the program returned value `0`. Thus, we enter `w` (*wrong*) in order to tell the debugger that the result was wrong. The debugging tool stores this choice as explained in Section 3. As the value of `exp` depends on several function calls on the right hand side of its definition, the tool now displays the first of these calls in a leftmost, innermost order:

```
fiblist 0 ~> _ : (_ : _)
```

The line above shows that the expression `fiblist 0` has been evaluated to a list that has at least two elements. This might be correct, but we are not too sure, since this result depends strongly on the evaluation context. A "don't know" in declarative debugging actually corresponds to the skipping of sub computations in the step-by-step mode, as described in the next section. We therefore press `s` (*skip*).

```
take 2 (_ : (_ : _)) ~> [_,_]
```

Actually, this looks quite good. By entering `c` (*correct*) we declare that this sub computation meets our expectation. Now the following calculation is displayed:

```
length [_,_] ~> 0
```

The function `length` is supposed to count the elements in a list. Since the argument is a two-element list, the result should be `2`, but it is actually `0`. By pressing `w` we therefore state that this calculation is erroneous. Now the debugger asks for the first sub computation leading to this result:

```
length [_]  ~> 0
```

This is also wrong, but for the sake of demonstration we delay our decision. By pressing the space bar (*step into*) we move to the sub expressions of `length [_]`. We now get to the final question:

```
length [] ~> 0
```

The length of an empty list `[]` is zero, so by pressing `c` (*correct*) we state that this evaluation step is correct. Now we have reached the end of the program execution, but a bug has not been isolated yet. We have narrowed down the error to the function call `length [_,_]`, but still there are unrated sub computations which might have contributed to the erroneous result. The tool asks if the user wants to restart the debugging session re-using previously given ratings:

```
end reached. press 'q' to abort or any other key to restart.
```

After pressing <SPACE>, the debugger restarts and asks for the remaining function calls. There is only one unrated call left within the erroneous sub computation:

```
length [_] ~> 0
```

Now we provide the rating we previously skipped. After entering `w` (*wrong*) it is evident which definition contains the error:

```
found bug in rule:
  lhs = length [_]
  rhs = 0
```

2.2 Step-by-Step Debugging and Virtual I/O

A further interesting advantage of our approach to reexecute the program with a strict evaluation strategy is the possibility to include "virtual I/O". During the execution of the original program, all externally defined I/O-actions with non-trivial results, i.e., other than `IO ()`, are stored in a special file. These values are retrieved during the debugging session. In addition, selected externally defined I/O-actions, e.g., `putChar`, are provided with a "virtual implementation". To show what this means, we demonstrate how the `main` action of the program found in Figure 2 is treated by our debugging tool. As described in the previous section, the program is executed to obtain the oracle in the file `IOExample.steps`. As this program contains user interaction, we also have to enter a line. We type `abc` for this demonstration. Meanwhile, along with the file containing the oracle, a file named `IOExample.ext` is written that contains the sequence of values for `getChar` and the number of bytes for their representation in the file.

```
~/oracle> cat IOExample.ext
3,'a'3,'b'3,'c'4,'\n'
```

```
module IOExample where

import Prelude hiding (getLine)

getLine :: IO String
getLine = getChar >>= testEOL

testEOL :: Char -> IO String
testEOL c = if c=='\n' then return []
                       else getLine >>= \ cs -> return (c:cs)

main = getLine >>= writeFile "userInput"
```

Fig. 2. I/O example

There is no need to identify the different calls to external functions, since I/O-actions will be executed in the strict version in exactly the same order as in the original program. This is of course essential for correctness. We now start the debugging tool, and look at the first two single steps by typing <SPACE> twice. This is what the tool displays at this point:

```
main
getLine
getLine ~> getChar >>= testEOL
main ~> (getChar >>= testEOL) >>= writeFile "userInput"
initial action computed. press any key to execute it
```

In step-by-step mode, we only get to see results when a subcomputation is finished. The above lines mean that the evaluation of both, `getLine` and `main` is now complete. The results are partial calls of the bind operator `(>>=)` waiting for the world, so to speak. We press an arbitrary key to start the action followed by a <SPACE> to make one more single step and get:

```
getChar >>= testEOL
getChar
```

When we hit <SPACE> now, two things happen at once. First, the value `'a'` is retrieved from the file and, second, a GUI called `B.I.O.tope` is started, which represents the virtual I/O environment. `B.I.O.tope` is told that someone has typed an `a` on the console, which is the "virtual I/O-action" we connected with `getChar`. The `B.I.O.tope` window is shown in Figure 3. Meanwhile, on the console we see the result of the call to `testEOL 'a'`, which we skip by typing `s`.

```
testEOL 'a' ~> (getChar >>= testEOL) >>= testEOL_lambda 'a'
(getChar >>= testEOL) >>= testEOL_lambda 'a'
```

Admittedly, the expression `testEOL_lambda 'a'` shows that the source code binding is improvable. Now we wonder, whether or not the current sub computation is interesting. We type `r` to have a look at the result and get:

```
(getChar >>= testEOL) >>= testEOL_lambda 'a' ~> IO "abc"
```

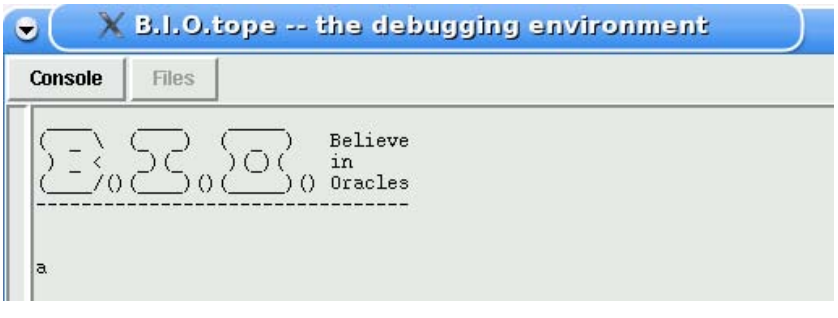


Fig. 3. The B.I.O.tope Virtual I/O Environment

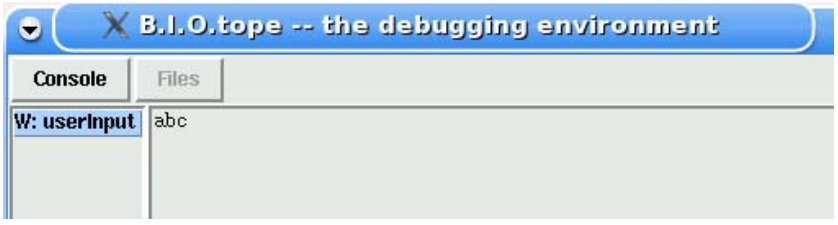


Fig. 4. Files in the B.I.O.tope Virtual I/O Environment

This is fine, so we decide to skip the computation by pressing `s`. Note, that as soon as a result is shown, we can also rate the sub computation, i.e., tell the tool that this result is correct or wrong. This information will then be considered if we restart the debugging session in inspect mode, cf. Section 2.1. It is also noteworthy that the virtual I/O commands are never issued twice, even if we had decided to go into the sub computation instead of skipping it.
The final action of our program is:

```
writeFile "userInput" "abc"
```

Executing this action brings another change to the `B.I.O.tope` as shown in Figure 4. There we can see the GUI has switched to the file dialog. It contains a list of files which have been read (`R:`) or written (`W:`) during the debugging session and clicking a file in this list makes the file contents visible as they are at the current point of the debugging session.

3 Implementation

In this section we present the runtime library `StrictSteps.hs` and its interaction with the programs generated by the transformation introduced in Section 2.1. Given the program in Figure 1, the transformation creates a Haskell program called `StrictExample.hs`. Each generated module uses the functions that are exported by the library `StrictSteps.hs`. It also imports the transformed versions of its original imports. In this case the only such module is `StrictPrelude`. Finally, some functions from the original Haskell `Prelude` are needed. The module head of `StrictExample.hs` looks like this:

```
module StrictExample where

import StrictSteps
import Prelude (Maybe(..),(.),Eq(..),Show(..),Ordering(..),
                Either(..),String,Bool(..),Char(..),Float(..))
import qualified Prelude (IO,return,(>>=))
import StrictPrelude
```

3.1 Encoding of Oracles

Conceptually, an oracle is a list of decisions. For each function call the oracle decides whether or not that call is evaluated. This list of decisions is being consumed while an instrumented program is evaluated according to a leftmost, innermost strategy. If the next entry of the oracle has value `True`, the next reduction step, according to strict evaluation order, will be evaluated. If the next entry has value `False`, then the next redex will be skipped and the result will be replaced by a placeholder value called `underscore`. In order to represent the oracle in a compact way, it is encoded as a list of natural numbers. A number n stands for a list of n entries whose value is `True` followed by one entry of value `False`. That the list is in this compact representation is an implementation detail which we hide by defining a stack with the usual interface:

```
type BoolStack = [Integer]
push :: BoolStack -> Bool -> BoolStack
pop  :: BoolStack -> (Bool, BoolStack)
```

The smallest oracle is the one that simply discards the whole expression given. Therefore an empty `BoolStack` is constructed according to the following declaration:

```
empty :: BoolStack
empty = [0]
```

Reconsider the example program in Figure 1. The oracle produced to evaluate `exp` was `[2,0,1,0,0,23]`. It represents a list containing 31 boolean entries: the first two entries have value `True`, then there are two `False` followed by one `True`, then three `False` and finally 23 `True`. Having value `True`, the first entry indicates that expression `exp` has to be evaluated. The next entry tells us that the next leftmost innermost call in Figure 1, i.e., `fiblist 0`, is also unfolded. The next two entries have value `False` and thereby indicate that the leftmost innermost calls `fib x` and `x+1` must not be evaluated but replaced by an `underscore` (also denoted by `_`) as a placeholder value. Replacing `x+1` by `_` the next call is `fibs _`. This call is then also evaluated whereas the next free expressions, `fib _`, `x+1` and another `fibs _` are discarded. The final 23 entries tell us that the remaining computation is totally strict (and 23 steps long).

3.2 The Representation of `underscore`

Expressions which are not evaluated are replaced by the special value `underscore`. Therefore, at least conceptually, every data type has to be augmented with a new element which represents that special value. From a semantical point of view, `underscore` resembles the undefined value $\bot$. If we were only interested in computing the same result with a strict semantics, we could actually represent `underscore` by a call to the Haskell function `undefined`, because discarded expressions are guaranteed to never be needed in the evaluation. However, we want the debugging tool to print intermediate results, therefore, we need to be able to distinguish undefined values from defined ones.

One option to implement this is to add a new constructor to every data declaration. However, this would make the inclusion of external functions and data types much more complicated. In order to, e.g., call `(+)`, we would need to convert to and from for each argument and the result. In addition, much of the behavior already provided would have to be duplicated, e.g., the way strings of characters are shown.

To avoid these further complications we make use of the fact that `underscore` is never evaluated outside the display routines. We represent `underscore` by an exception.

```
underscore :: a
underscore = throw NonTermination
```

Since the oracle that guides the evaluation indicates which expressions are needed to run the program, it is guaranteed that this value will never be accessed while the program is being evaluated. It is considered only to print intermediate results. As printing is an I/O-action the exception can be safely caught whenever undefined values are processed.

3.3 Representing Values

The debugger must be able to display a textual representation of the arguments and results of function calls. In order to provide more flexibility for the debugging tool, we represent expressions in a term structure rather than a simple string. This makes it possible to, e.g., restrict the depth in which expressions are shown and enables pretty printing. Therefore, the data type `Term` is introduced:

```
data Term = Term String [Term] | Underscore | Fail String
```

The constructor `Term` contains the name of the applied symbol and a term representation of its arguments. As discussed above, `Underscore` stands for those expressions, which were not evaluated. Finally, `Fail` represents exceptions that occurred during the execution along with an error message. The implementation of the corresponding mechanism is, however, not finished.

In order to retrieve term representations of a given expression, each data type has to be an instance of the class `ShowTerm`:

```
class ShowTerm a where
  showCons :: a -> DebugMonad Term
```

These instances are automatically generated by the transformation. For example, the following instance declaration is generated for lists:

```
instance ShowTerm a => ShowTerm [a] where
  showCons [] = return (Term "[]" [])
  showCons (x1:x2) = do sx1 <- showTerm x1
                        sx2 <- showTerm x2
                        return (Term ":" [sx1,sx2])
```

The generated instances depend on the generic function `showTerm`, which is responsible for catching the exception thrown by `underscore`:

```
showTerm :: ShowTerm a => a -> DebugMonad Term
showTerm x = liftIO (catch (x `seq` return Nothing) (return . Just)) >>=
             maybe (showCons x) (\ e -> case e of
                                  NonTermination -> return Underscore
                                  ErrorCall s    -> return (Fail s))
```

3.4 The Debug Monad

The whole evaluation of the generated program takes place in a monad, the `DebugMonad`. This monad is a state monad, managing the following state:

```
data DebuggerState = DebuggerState { oracle      :: BoolStack,
                                     displayMode :: IORef DMode,
                                     skipped     :: BoolStack,
                                     unrated     :: BoolStack,
                                     stepMode    :: StepMode }
```

`oracle` contains the part of the oracle that has not yet been consumed.

`displayMode` contains display options like the verbosity level and the depth up to which terms should be printed. In addition this field contains a flag indicating which of the two debugging modes described in Section 2 is active.

`skipped`, `unrated` are needed to implement declarative debugging and contain the information about whether or not a given sub computation has been marked as *correct* by the user. Like oracles, `skipped` and `unrated` contain a series of boolean information about expressions occurring in the computation. They are therefore also implemented as values of type `BoolStack`. Both stacks contain a `False` for each expression that has been rated as correct and a `True` for each expression that has not yet been rated by the user. The difference between both stacks is that `skipped` holds the ratings of the functions that have already been displayed in the current program run, whereas `unrated` holds the ratings of the function calls that have not yet been displayed in the current program run. Whenever the end of the computation is reached without isolating an error, `unrated` becomes `skipped`, `skipped` is reinitialized as an empty `BoolStack`, and the debugging session restarts, cf. Section 3.5 for more details.

In addition to the parts of `DebugState` described so far, there are three *evaluation modes* encoded in data type `StepMode`. In the next subsection these modes will be described in detail:

```
data StepMode = StepBackground | StepInteractive | StepCorrect
```

The monad `DebugMonad a` is employed to manage the debugger's internal state while evaluating an expression that has result type `a`.

```
type DebugMonad a
  = StateT DebuggerState (ErrorT (Maybe BugReport) IO) a
```

Values of type `BugReport` are used to report an erroneous program rule which is represented by two constructor terms, i.e., the call along with the arguments and the result. The result delivered by the debugger is lifted into monad `IO`, because the debugger has to interact with the user via I/O actions while it evaluates the expression. This monad is transformed by the monad transformer `ErrorT`, so that not only the result can be returned, but also the evaluation can be truncated reporting the location of an error (`Just bug`) or indicating that the user has aborted the debugging session (`Nothing`). As soon as a program error has been pinned down to a single program rule, the evaluation is truncated and that program rule is reported to the user. One step further, this monad is transformed by the monad transformer `StateT` in order to let the debugger read and write its state while executing a program.

Manipulation of the oracle within the debug monad's state is done by the `eval` function which consumes one entry from the current oracle. Depending on the value of this entry, it either evaluates its argument and returns the result of this evaluation, or it refrains from evaluating its argument and returns `underscore` as a placeholder.

```
eval :: ShowTerm a => DebugMonad a -> DebugMonad a
eval a = do state <- get
            let (orc, needed) = pop (oracle state)
            put (state {oracle = orc})
            if needed then a else return underscore
```

All functions are transformed to monadic actions of type `DebugMonad`. For example, the types of the transformed versions of the functions in Figure 1 are:

```
length :: ShowTerm a => [a] -> DebugMonad Int
exp :: DebugMonad Int
fiblist :: Int -> DebugMonad [Int]
fib :: Int -> DebugMonad Int
```

In the original program `exp` is a value of type `Int`, but in the transformed program every evaluation takes place in the debug monad. The resulting function may return a value of type `Int` or abort yielding a bug report. Similarly, the transformed version of `length` is still a function taking two arguments, but now it yields an operation that has to be executed in the debug monad to retrieve its result.

The transformation also adds a function `main`, an action of type `IO ()`. This action initializes the debugging session by loading the oracle from disk and starting the interactive debug session for the given expression.

Before we can explain some details of how function declarations are transformed, we need to introduce the function that is at the heart of user interaction with the debugger, called `traceFunCall`.

3.5 User Interaction with `traceFunCall`

The function `traceFunCall` interfaces the instrumented program with the interactive debugger. Every top level declaration is augmented with a call to this function, which has the following type signature:

```
traceFunCall :: ShowTerm a =>
                DebugMonad Term -> DebugMonad a -> DebugMonad a
```

The first argument contains an action to retrieve the term representation of the function call about to be evaluated, cf. Section 3.3. The second argument is the function body that has been lifted into the debugging monad as described in the next section. The class context `ShowTerm a =>` makes sure that the result can be displayed to the user. Apart from printing information for the user and waiting for his input, `traceFunCall` is responsible for correctly manipulating the two boolean stacks, `skipped` and `unrated`, which are part of the monad state. The way in which these stacks are manipulated depends on the evaluation mode, which is also part of the debug monad's state:

```
traceFunCall term expr = do
  st <- get
  case stepMode st of
```

Depending on the mode, `traceFunCall` shows one of the following behaviors:

Mode `StepCorrect` indicates that the computation currently executed was rated by the user as *correct*. According to the declarative debugging technique, all its sub computations are also considered to be correct. Making use of this aspect of declarative debugging, the two stacks `skipped` and `unrated` are implemented such that they do not contain any information about nested correct sub computations at all. Therefore, `skipped` and `unrated` remain unchanged when a sub expression is evaluated with mode `StepCorrect`:

```
    StepCorrect -> eval expr
```

Mode `StepBackground`. This mode indicates that only the result of the current computation should be computed. During the computation of this result there should be no interaction with the user. Remember that the stacks `skipped` and `unrated` should contain a `True` for every expression that was not yet marked as correct by the user, cf. Section 3.4. The stack `skipped` contains the information about the past computation while `unrated` contains information about the remaining computation. This invariant is maintained by popping one entry from `unrated` and pushing it to `skipped` for every evaluation step in mode `StepBackground`. Since this operation occurs in more than one place, we defined the operation `restack`:

```
restack :: DebugMonad Bool
restack = do
```

```
st <- get
let (isUnrated,newUnrated) = pop (unrated st)
put st{unrated=newUnrated,
       skipped=push isUnrated (skipped st)}
return isUnrated
```

If the entry popped from `unrated` indicates that the computation was rated correct in a previous iteration of the debugging cycle, the evaluation mode is then switched to `StepCorrect`.

```
StepBackground -> do
  isUnrated <- restack
  unless isUnrated (setStepMode StepCorrect)
  eval expr
```

Mode `StepInteractive`. All interaction between the user and the debugger takes place in mode `StepInteractive`. User interaction will only take place when the current expression has not yet been rated in a previous iteration of the debugging cycle. Therefore, the first thing to do is inspect the top value of the stack `unrated` while remembering the current state for eventual future reference. If that value is `False` the mode is changed to `StepCorrect` and the expression is evaluated.

```
StepInteractive -> do
  st <- get
  isUnrated <- restack
  if isUnrated then do setStepMode StepCorrect
                       eval expr
```

Otherwise the current sub computation is executed in mode `StepBackground`. After the sub computation is finished the resulting value is displayed and the user is asked whether it is correct.

```
    else do setStepMode StepBackground
            result <- eval expr
            userAnswer <- askUserToRate result
            case userAnswer of
```

There are four possibilities for the user to answer:

1. If the user marks the displayed result as correct, the new value of stack `skipped` is reset to the old value with an additional `False`. As described above, the `False` indicates that the expression was rated as correct. Furthermore we do not have to keep track of the details of correct sub computations and can therefore forget about the information in the current value of `skipped`. Then `traceFunCall` returns the computed result.

```
Correct -> do setSkipped (push False (skipped st))
              return result
```

2. If the user skips the current function call `traceFunCall` simply returns the computed result.

```
Skip -> return result
```

3. If the user is not yet sure about the computed results and wants to inspect the details of the sub computation before rating correct or wrong, he can *step into* the sub computation. In this case the old state has to be reset relocating the top value of stack `unrated` to `skipped`.

```
StepInto -> do put st
               restack
               eval expr
```

4. If the user has rated the resulting value as wrong, the debugging session will confine itself to searching for the bug in the current expression. If it finds a bug in one of them, then in turn it restricts itself to searching for the bug in that sub expression. If it does not find a bug in any of its sub expressions, it is clear that the definition of the currently called function is erroneous, and the current function call along with its result is reported as the result of the debugging session.

```
Wrong -> do put st
            debug expr
            throw (bugReport term result)
```

If the debugging tool is in step-by-step mode, cf. Section 2.2, `traceFunCall` will not calculate the result of the current function call until the user requests it. Instead it starts by displaying only the function call and giving the user an opportunity to move forward to rating its subexpressions without having to evaluate the expression first.

3.6 Transforming Function Bodies

After having explained all the operations occurring in the transformed programs, the actual transformation of function bodies is simple. Rather than giving a formal description of the transformation we prefer to explain the procedure by example. Reconsider the definition of function `fiblist` of Figure 1:

```
fiblist x = fib x : fibs (x+1)
```

This function is transformed to the following definition:

```
fiblist ::  Int -> DebugMonad [Int]
fiblist x1 = traceFunCall (do sx1 <- showTerm x1
                              return (Term "fiblist" [sx1]))
                          (fib x1 >>= \x4 ->
                           x1 + 1 >>= \x2 ->
                           fiblist x2 >>= \x3 ->
                           return (x4 : x3))
```

As explained above, each step of the evaluation is controlled by the operation `traceFunCall`. This operation takes two arguments. The first argument is an action to retrieve a representation of the current expression corresponding to the explanations of Section 3.3. The second argument is the monadic action to compute the current expression. Note that this monadic action is constructed in accordance to leftmost, innermost evaluation: first the call to `fib` is evaluated then `(+)` and finally `fiblist`. The oracle information does not consider constructor calls, so the result is constructed as `(x4 : x3)`.

3.7 Higher Order Functions

Up to now, our debugger supports higher order functions by displaying partial applications. The extension to more sophisticated representations by, e.g., collecting a function graph as proposed in [3] seems possible but has not been implemented yet.

Representing partial applications is not a conceptual extension. The main problem is to formulate a working instance of class `ShowTerm`. Primitive data types like `a -> b` are augmented with a wrapping constructor that holds, in addition to the actual value, a term representation.

```
data Prim a = Prim Term a
```

The `ShowTerm` instance is uniformly defined for all primitive types as:

```
instance showTerm (Prim a) where
  showCons (Prim x _) = return x
```

Because of the transformation to the `DebugMonad` and the wrapping constructor `Prim` the higher order function

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

is transformed to a function of type

```
foldr :: (ShowTerm a,ShowTerm b) =>
         Prim (a -> DebugMonad (Prim (b -> DebugMonad b))) ->
         b -> [a] -> DebugMonad b
```

and the body of `foldr` is transformed to

```
foldr x1 x2 x3 = traceFunCall
  (do sx1 <- showTerm x1
      sx2 <- showTerm x2
      sx3 <- showTerm x3
      return (Term "foldr" [x1,x2,x3]))
  (eval (case x3 of
        [] -> return x2
        x4:x5 -> do x7 <- apply x1 x4
                    x6 <- foldr x1 x2 x5
                    apply x7 x6))
```

Note that the (implicit) applications `(f x (...))` of the original definition have been transformed to the explicit calls to an `apply` function that is defined as follows:

```
apply :: Prim (a -> DebugMonad b) -> a -> DebugMonad b
apply (Prim _ f) x = f x
```

The partial calls corresponding to `apply` are introduced by functions `pc`n where n is the arity of the partially applied function. For example, the expression `(foldr (+) 0 [])` is transformed to

```
foldr (pc2 (Term "+" []) (+)) 0 []
```

and the function `pc2`, responsible for partial calls of arity two, is defined as:

```
pc2 :: ShowTerm a => Term -> (a -> b -> DebugMonad res) ->
       Prim (a -> DebugMonad (Prim (b -> DebugMonad res))
pc2 (Term n xs) f =
  Prim (c []) (\ x -> do sx <- showTerm x
                         return (Prim (c [sx]) (f x)))
  where c = Term n . (xs++)
```

The resulting program needs to contain functions `pc`n for all occurring arities. Although we have some experience expressing such similar applications by employing a small set of combinators, we were not able to do so in this application. The special problem here in comparison to other applications of a combinator approach is the flow of information. Here we have information going from the arguments to the call (the argument's term representation) and also from the call to the arguments (the wrapping with the `Prim` constructor). Typical problems that can be solved with combinators have only either way of information flow.

4 Comparison with Related Work

As discussed in Section 1, the state of the art tools to compare with is HAT [2]. However, a thorough benchmarking comparison is not without difficulties. HAT was developed for Haskell while b.i.o. is a debugging tool for Curry. Although the semantics of both languages coincide for purely functional programs, there are still noteworthy differences. For example, the arithmetic used in the Curry implementation which b.i.o. is part of is defined on pure, i.e. not external, data structures. In order to be as comparable as possible we have ported two Haskell programs, the standard Fibonacci example and one from the NoFib Haskell Benchmark Suite, to Curry. With our Curry compiler we have transformed those programs into an intermediate language. This intermediate language is the usual starting point for the transformations to produce the oracle and the strict debugging monad described in Section 3. In addition we have retranslated the intermediate language to Haskell and used Hat on the resulting programs. All benchmarks represent the average of ten runs and were done on

Table 1. Benchmark for Trace Recording

Program	no tracing	HAT	b.i.o.
Run-Time `Fibonacci`	0.18 s	0.8 s	0.3 s
Size of Trace `Fibonacci`	–	14.7 MB	8 B
Run-Time `queens 9`	2.11 s	30 s	8 s
Size of Trace `queens 9`	–	250 MB	1.3 MB

a local file system on an AMD Athlon(TM) machine with 1.33Mhz speed and 512MB memory. Table 1 shows the resulting time and space requirements.

The actual usage of the systems to search for bugs is even harder to compare. The maximal time the user has to wait in our system while inspecting the above example programs is 9 seconds for `Fibonacci` and 3 minutes for `queens 9`. This is the time it takes the strict version to execute the whole program. This time is not very expressive for practice, however, as the user typically inspects intermediate computations. By design, the maximal time is the sum of the time of all intermediate computations, such that inspecting any sub computation is considerably quicker. Therefore, we have no idea how to design an objective benchmark to compare the actual usage of our system with that of HAT.

5 Conclusion and Future Work

We have presented the usage and implementation of a debugging tool utilizing the oracle technique developed in [1]. Up to now, the debugger features a declarative debugging mode as well as a step-by-step mode corresponding to a leftmost, innermost evaluation strategy. In addition, a virtual I/O environment gives the user the opportunity to see side effects issued by the program. Up to now, this environment features console output and file access. An extension to other often used I/O actions like `IORef`s and sockets is straight forward. In extension to the system as presented at the conference we have added the possibility to designate *trusted functions*. We cannot omit transforming trusted functions neither to produce nor to consume the oracle. But executing trusted function calls in the strict debug monad is considerably faster, and the user is not bothered with any details of that execution. Higher order applications of untrusted functions are, however, displayed as usual even if that application took place inside the execution of a trusted function.

The main limitation of our approach, in its current state, is the lack of treating run-time errors. Improving this is clearly important for debugging purposes.

Other room for improvement is of course adding to the list of debugging features. Many useful techniques are easy to integrate into the framework like spy points, observations and remembering questions already asked. We plan to include some of the features described in [7] as well as those provided by HAT [2], as far as they fit into the framework.

References

1. Braßel, B., Fischer, S., Hanus, M., Huch, F., Vidal, G.: Lazy call-by-value evaluation. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), pp. 265–276 (2007)
2. Chitil, O., Runciman, C., Wallace, M.: Freja, hat and hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 176–193. Springer, Heidelberg (2001)
3. Chitil, O., Davie, T.: Display of functional values for debugging. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 326–337. Springer, Heidelberg (2007)
4. Gill, A.: Debugging Haskell by observing intermediate datastructures. Electronic Notes in Theoretical Computer Science 41(1) (2001)
5. Nilsson, H., Sparud, J.: The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. Automated Software Engineering 4(2), 121–150 (1997)
6. Pope, B.: Declarative Debugging with Buddha. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 273–308. Springer, Heidelberg (2005)
7. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
8. Sparud, J., Runciman, C.: Tracing Lazy Functional Computations Using Redex Trails. In: Hartel, P.H., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 291–308. Springer, Heidelberg (1997)

Uniqueness Typing Simplified

Edsko de Vries[1,⋆], Rinus Plasmeijer[2], and David M. Abrahamson[1]

[1] Trinity College Dublin, Ireland
{devriese,david}@cs.tcd.ie
[2] Radboud Universiteit Nijmegen, Netherlands
rinus@cs.ru.nl

Abstract. We present a uniqueness type system that is simpler than both Clean's uniqueness system and a system we proposed previously. The new type system is straightforward to implement and add to existing compilers, and can easily be extended with advanced features such as higher rank types and impredicativity. We describe our implementation in *Morrow*, an experimental functional language with both these features. Finally, we prove soundness of the core type system with respect to the call-by-need lambda calculus.

1 Introduction to Uniqueness Typing

An important property of pure functional programming languages is *referential transparency*: the same expression used twice must have the same value twice. This makes equational reasoning possible and aids program analysis, but most languages do not have this property. For example, in the following C fragment,

```
int f(FILE* file) {
  int a = fgetc(file); // Read a character from 'file'
  int b = fgetc(file);
  return a + b;
}
```

it is understood that `a` and `b` can have different values, even though we are applying the same function (`fgetc`) to the same input (`file`). Although the input is syntactically identical, the structure denoted by `file` is modified by each call to `fgetc` (the file pointer is advanced)—`fgetc` has a side effect.

In this example there would be no problem with referential transparency *if there was only a single reference to* `file`. A side effect on a variable (`file`) is okay as long as that variable is never used again: it is okay for a function to modify its input if the input is not *shared*. Referential transparency then trivially holds because the same expression never occurs more than once.

Uniqueness typing takes advantage of this observation to add side effects to a functional language without sacrificing referential transparency. The same function `f` implemented in a functional language using uniqueness typing gives

⋆ Supported by the Irish Research Council for Science, Engineering and Technology.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 201–218, 2008.

```
f file0 = let (a, file1) = fgetc file0
              (b, file2) = fgetc file1
          in (a + b, file2)
```

Rather than just returning the read character, `fgetc` returns a pair consisting of the read character *and a new file*, `file1`. Although `file0` and `file1` point to the same file on disk, they are conceptually *and* syntactically different, and thus it is clear that a and b may have different values. The uniqueness type system guarantees that `fgetc` is never applied to an argument which is used again (shared). For example, the type checker would reject

```
f file0 = let (a, file1) = fgetc file0
              (b, file2) = fgetc file0
          in (a + b, file0)
```

Sharing information is recorded as an attribute on the type of a term. This attribute is either $\bullet$ for unique (guaranteed not to be shared) or $\times$ for non-unique (may or may not be shared). For instance, $\texttt{File}^{\bullet}$ is the type of files that are guaranteed not to be shared, and the type of `fgetc` might be

$$\texttt{fgetc} :: \texttt{File}^{\bullet} \xrightarrow{\times} (\texttt{Char}^{\times}, \texttt{File}^{u})^{v}$$

The attribute on the arrow means that the function `fgetc` *itself* is non-unique (Sect. 4.2). The uniqueness variable u on the result means that it is up to the programmer to decide if they want to treat it as unique or shared (Sect. 6).

2 Contributions of This Paper

The type system we present in this paper is based on that of the programming language *Clean* [1,2]. However, Clean's type system has a number of drawbacks.

- Types and attributes are regarded as two different entities, which limits expressiveness and impedes adding uniqueness typing to existing compilers.
- Types often involve implications between uniqueness attributes. For example, the function `const` has type

  $$\texttt{const} :: t^{u} \xrightarrow{\times} s^{v} \xrightarrow{w} t^{u}, [w \leq u]$$
  ```
  const x y = x
  ```

  The constraint $[w \leq u]$ denotes that if u is unique, then w must be unique (u implies w).[1] The need for this constraint will be explained in Sect. 4.2, but the presence of these constraints complicates the work of the type checker (the heart of the typechecker is a unification algorithm, and unification cannot deal with inequalities) and makes extending the type system to support modern features such as arbitrary rank types difficult.

[1] Perhaps the choice of the symbol $\leq$ is unfortunate. In logic $a \leq b$ denotes a implies b, whereas here $u \leq v$ denotes v implies u. Usage here conforms to *Clean* conventions.

– Clean distinguishes between non-unique terms, unique terms (which are unique now but may become non-unique later), and *necessarily unique* terms (which must remain unique forever). Moreover, Clean's type system has a subtyping relation between unique and non-unique terms. Both these features make the type system unnecessarily complicated.

In this paper, we make the following contributions.

– Section 3 shows that we can regard uniqueness attributes as type constructors of a special kind. This increases the expressive power of the type system and simplifies the presentation and implementation of uniqueness typing.
– Section 4 presents the type system proper and shows how to avoid inequality constraints by allowing arbitrary boolean expressions as uniqueness attributes. This facilitates extending the type system with advanced features and enables the use of unification to solve relations between attributes.
– Section 6 shows how to avoid subtyping. We argued a similar point in a previous paper [3] but unfortunately the approach in that paper requires a second uniqueness attribute on the function arrow, offsetting the advantage of removing subtyping. Our new approach does not have this disadvantage.
– Section 7 describes our implementation in *Morrow*. Morrow supports higher rank types and impredicativity, but adding support for uniqueness typing to Morrow required only a few changes to the compiler. This provides strong evidence for our claim that retrofitting uniqueness typing to an existing compiler, and extending uniqueness typing with advanced features, becomes straightforward using the techniques in this paper. As far as the authors are aware, this is also the first substructural type system to have these features.
– Finally, we prove soundness of our type system in Sect. 8 with respect to the call-by-need lambda calculus [4].

3 Attributes Are Types

In this section, we show that we can regard types and attributes as one syntactic category. This simplifies both the presentation and implementation of a uniqueness type system and increases the expressive power of the type system.

If we regard types and attributes as distinct, we need type variables and attribute variables, and we need quantification ($\forall$) over type variables and attribute variables. In addition, the status of arguments to algebraic datatypes (such as `List a`) is unclear: are they types, attributes, or types *with* an attribute?

These issues are clarified when we regard types and attributes as a single syntactic category. Thus `Int` and `Bool` are types, and so are $\bullet$ (unique) and $\times$ (non-unique). We regard $\texttt{Int}^{\times}$ as syntactic sugar for the application of a special type constructor `Attr` to two arguments, `Int` and $\times$. There are no values of type $\times$, nor are there values of type `Int`, because `Int` is lacking a uniqueness attribute (there are however values of type $\texttt{Int}^{\times}$).

Types that do not classify values are not a new concept. For example, they arise in Haskell as *type constructors* such as the list type constructor (`[]`). We

Kind language

κ	::=	kind
	$\mathcal{T}$	base type
	$\mathcal{U}$	uniqueness attribute
	$*$	base type together with a uniqueness attribute
	$\kappa_1 \to \kappa_2$	type constructors

Type constants

`Int`, `Bool`	:: $\mathcal{T}$	base type
$\to$	:: $* \to * \to \mathcal{T}$	function space
$\bullet$, $\times$	:: $\mathcal{U}$	unique, non-unique
$\vee$, $\wedge$	:: $\mathcal{U} \to \mathcal{U} \to \mathcal{U}$	disjunction, conjunction
$\neg$	:: $\mathcal{U} \to \mathcal{U}$	negation
`Attr`	:: $\mathcal{T} \to \mathcal{U} \to *$	combine a base type and attribute

Syntactic conventions

t^u	$\equiv$ `Attr` $t\ u$
$a \xrightarrow{u} b$	$\equiv$ `Attr` $(a \to b)\ u$

Fig. 1. The kind language and some type constructors with their kinds

can make precise which types do and do not classify values by introducing a kind system [5]. Kinds can be regarded as the "types of types". By definition, the kind of types that classify values is denoted by $*$. In Haskell, we have `Int :: *`, `Bool :: *`, but `[] :: * → *`. The idea of letting the language of vanilla types and additional properties coincide is not new either (e.g., [6,7]), but as far as the authors are aware it is new in the context of substructural type systems.

Since we do not regard `Int` as a type classifying values, its kind cannot be $*$ in our type system. Instead, we introduce two new kinds, $\mathcal{T}$ and $\mathcal{U}$, classifying "base types" and uniqueness attributes. Since `Attr` combines a base type and an attribute into a type of kind $*$, its kind is $\mathcal{T} \to \mathcal{U} \to *$. The kind language and some type constructors along with their kinds are listed in Fig. 1. At this point it is useful to introduce the following convention.

> **(Syntactic convention.)** Type variables[2] of kind $\mathcal{T}$ and $\mathcal{U}$ will be denoted by t, s and u, v. Type variables of kind $*$ will be denoted by a, b.

One advantage of treating attributes as types is that we can use type variables to range over base types, uniqueness attributes or types with an attribute, simply by varying the kind of the type variable. This gives more expressive power when defining algebraic data types. For example, we can define:

```
newtype X a = MkX a
newtype Y t = MkY t^×
newtype Z u = MkZ Int^u
```

[2] Strictly speaking, these are meta variables, not object language type variables. Our core language does not include universal quantification.

$e ::=$		expression	$\tau_k ::=$		type
	$x^{\odot}$	variable (used once)		c_k	constant
	$x^{\otimes}$	variable (used more than once)		$\tau_{(k' \to k)}\ \tau_{k'}$	type application
	$\lambda x \cdot e$	abstraction			
	$e\,e$	application			

Fig. 2. Expression and type language for the core system

The type of a constructor argument must have kind $*$; hence, the first datatype is parameterized by an attributed type (a type of kind $*$), the second by a base type (a type of kind $\mathcal{T}$), and the third by a uniqueness attribute (a type of kind $\mathcal{U}$). The kinds of `X`, `Y` and `Z` are therefore $* \to \mathcal{T}$, $\mathcal{T} \to \mathcal{T}$ and $\mathcal{U} \to \mathcal{T}$, respectively. The codomain is $\mathcal{T}$ in all cases, since `X Int`$^{\times}$ still lacks an attribute; `(X Int`$^{\times}$`)`$^{\bullet}$ on the other hand is a unique `X` containing a non-unique `Int`. So, assuming $(5 :: \mathrm{Int}^{\times})$, we have $(\texttt{MkX}\ 5 :: \texttt{X}^{u}\ \texttt{Int}^{\times})$, $(\texttt{MkY}\ 5 :: \texttt{Y}^{u}\ \texttt{Int})$ and $(\texttt{MkZ}\ 5 :: \texttt{Z}^{u}\ \times)$.

In Clean, we can only define the first of these three datatypes, so we have gained expressive power. What is more, although we have used syntactic conventions to give a visual clue about the kinds of the type variables, the kinds of these types can automatically be inferred by the kind checker, so the expressive power comes at no cost to the programmer.

There are two possible variations to the kind system we propose. We could treat `Int`$^{\times}$ as the application of $(\texttt{Int} :: \mathcal{U} \to *)$ to $(\times :: \mathcal{U})$, or as the (postfix) application of $(\times :: \mathcal{T} \to *)$ to $(\texttt{Int} :: \mathcal{T})$, avoiding the need for `Attr`. We prefer distinguishing between $\mathcal{T}$, $\mathcal{U}$ and $*$, but if the reader feels otherwise they should feel free to read $\mathcal{T}$ as syntactic sugar for $(\mathcal{U} \to *)$, or $\mathcal{U}$ as syntactic sugar for $(\mathcal{T} \to *)$. In all three variations only types of kind $*$ are inhabited, as usual.

4 Removing Constraints

In this section we show that by allowing arbitrary boolean expressions[3] as uniqueness attributes (reading "true" for unique and "false" for non-unique) we can recode implications between uniqueness attributes as equalities. This makes the type system so similar to the classical Hindley/Milner type system that standard type inference algorithms can be applied and modern extensions such as arbitrary rank types can be incorporated without much difficulty.

The expression language and type language are defined in Fig. 2 (types have been indexed by their kind k). Both are almost entirely standard, except that we assume that a sharing analysis has annotated variable uses with $\odot$ or $\otimes$. A variable x marked as $x^{\odot}$ is used only once within its scope; a variable marked as $x^{\otimes}$ is used more than once. The typing rules are listed in Fig. 3. The typing relation takes the form

$$\Gamma \vdash e : \tau|_{fv}$$

[3] Although the typing rules only use disjunctions between uniqueness attributes, more complicated expressions can be introduced when unifying two boolean expressions.

$$\frac{}{\Gamma, x : t^u \vdash x^{\odot} : t^u|_{x:u}}\ \textsc{Var}^{\odot} \qquad \frac{}{\Gamma, x : t^{\times} \vdash x^{\otimes} : t^{\times}|_{x:\times}}\ \textsc{Var}^{\otimes}$$

$$\frac{\Gamma, x : a \vdash e : b|_{fv} \qquad fv' = \triangleright_x\, fv}{\Gamma \vdash \lambda x \cdot e : a \xrightarrow{\bigvee fv'} b|_{fv'}}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : a \xrightarrow{u} b|_{fv_1} \qquad \Gamma \vdash e_2 : a|_{fv_2}}{\Gamma \vdash e_1\, e_2 : b|_{fv_1 \cup fv_2}}\ \textsc{App}$$

Fig. 3. Typing rules for the core lambda calculus

which reads as "in environment Γ, expression e has type τ; the attributes on the types of the free variables in e are fv". Both Γ and fv are mappings from term variables to types; the only difference is that Γ maps variables to types of kind $*$ and fv maps variables to types of kind $\mathcal{U}$ (in other words, to uniqueness attributes). The typing rule for abstraction uses fv to determine whether a function needs to be unique (this is discussed in more detail in Sect. 4.2).

The rules are similar to the Hindley/Milner rules, except that they maintain some extra information about uniqueness. The underlying base system is unchanged, so that uniqueness typing can be seen as an "add-on".

4.1 Variables

We need to distinguish variables that are used once in their scope and variables that are used multiple times. The rule for variables that are used only once ($\textsc{Var}^{\odot}$) is identical to the normal Hindley/Milder rule, and we simply look up the type of the variable in the environment. Note that even when a variable is used only once, that does not automatically make its type unique. For example, there is only one use of `x` in the identity function:

```
id x = x⊙
```

but when a shared term is passed to `id`, it will still be shared when it is returned from `id`. On the other hand, if a variable is used more than once (rule $\textsc{Var}^{\otimes}$), its type must be non-unique (shared).

4.2 Partial Application

Dealing correctly with partial application is probably the most subtle aspect of uniqueness typing. We will demonstrate the problem using a simple example. Temporarily ignoring the attributes on arrows, the type of `dup` is

```
dup :: t× → (t×, t×)u
dup x = (x⊗, x⊗)
```

Since `dup` duplicates its argument, it only accepts non-unique arguments. The type checker can easily recognize that `dup` duplicates x because there is more than one use of x in the function body, which is therefore marked as $\otimes$. However, what if we rewrite `dup` as

```
dup' x = (\f -> (f⊗ ⊥, f⊗ ⊥)) (const x⊙)
```

Now there is only one reference to x, which is therefore marked as $\odot$. Still ignoring the attributes on arrows, the function `const` is defined as

```
const :: t^u → s^v → t^u
const x y = x
```

It would therefore seem that the type of `dup'` is

```
dup' :: t^u → (t^u, t^u)^v
```

But that cannot be correct, because this type of `dup'` tells us that if we pass a single unique t to `dup'`, it will return a pair of two unique ts. However, the full type of `const` in our type system is

```
const :: t^u -×→ s^v -u→ t^u
```

If you pass in a unique t, you get a *unique* function from s to t: a function that can only be used once. Conversely, *if* you use a partial application of `const` more than once, the argument to `const` must be non-unique. The type of `dup'` is therefore

```
dup' :: t^× -×→ (t^×, t^×)^u
```

Reassuringly, this is the same type as the type of `dup`. In general, a function must be unique (and can be applied only once) if it has any unique elements in its closure (the environment that binds the free variables in the function body).

4.3 Abstraction and Application

The rule for abstractions is the same as the Hindley/Milner rule, except that we must determine the value of the attribute on the arrow. As discussed in Sect. 4.2, a function must be unique if it has any unique elements in its closure. The closure of a function $\lambda x \cdot e$ consists of the free variables in the body e of the function, minus x. The attributes on the free variables in the body of the function are recorded in fv; using $fv' = \triangleright_x fv$ (domain subtraction) to denote fv with x removed from its domain, we use the disjunction $\bigvee fv'$ of all the attributes in the range of fv' as the uniqueness attribute on the arrow (recall that we treat uniqueness attributes as boolean expressions).

The rule for application is the normal one, except that we collect the free variables. The attribute on the arrow is ignored (we can apply both unique and shared functions).

4.4 Encoding Constraints

In general, we can always recode a type of the form

$$\ldots \square^u \quad \ldots \square^v \ldots, [u \leq v]$$

using a disjunction

$$\ldots \square^{u \vee v} \ldots \square^v \ldots$$

This faithfully models the implication: when v is unique, $u \vee v$ reduces to unique, but when v is non-unique, $u \vee v$ reduces to u. For example, in Clean the function `fst` that extracts the first element of a pair has the type

```
fst :: (t^u, s^v)^w → t^u, [w ≤ u]
fst (x, y) = x
```

which we can recode as

```
fst :: (t^u, s^v)^(w∨u) → t^u
```

However, in many cases we can do slightly better. For example, suppose the typing rule for pairs is

$$\frac{\Gamma \vdash e_1 : a|_{fv} \qquad \Gamma \vdash e_2 : b|_{fv}}{\Gamma \vdash (e_1, e_2) : (a, b)^u|_{fv}} \quad \textsc{Pair}$$

then for every derivation of $e :: (a, b)^\bullet$, there is also a derivation of $e :: (a, b)^\times$ (because the typing rule leaves the attribute on the pair free). That means that we can simplify the type of `fst` to

```
fst' :: (t^u, s^v)^u → t^u
```

The only pairs accepted by `fst` but rejected by `fst'` are unique pairs, but since the type checker will never infer a pair to be unique (but always either non-unique or polymorphic in its uniqueness), that situation will never arise. We took advantage of the same principle in the rule for abstraction, where we recoded a type

$$\ldots \xrightarrow{u} \ldots, [u \leq v, u \leq w, \ldots]$$

as

$$\ldots \xrightarrow{v \vee w \vee \ldots} \ldots$$

This will force some functions to be non-unique which would otherwise be polymorphic in their uniqueness, but that cannot cause any type errors: the rule for function application ignores the uniqueness attribute on the function, and non-unique functions can be used multiple times.

5 Boolean Unification

One advantage of removing constraints from the type language is that standard inference algorithms (such as algorithm $\mathcal{W}$ [8]) can be applied without any modifications. The inference algorithm will depend on a unification algorithm, which must be modified to use boolean unification when unifying two terms of kind $\mathcal{U}$. Suppose we have two terms g and h

$$g :: t^\bullet \xrightarrow{\times} \ldots \qquad h :: t^{u \vee v}$$

Should the application $g\,h$ be allowed? If so, we must be able to unify $u \vee v \simeq \bullet$. This equation has many solutions, such as $[u \mapsto \bullet, v \mapsto v]$, $[u \mapsto u, v \mapsto \bullet]$, or

```
unify₀ :: BooleanAlgebra a => a -> [Var] -> (Subst a, a)
unify₀ t [] = ([], t)
unify₀ t (x : xs) = (s_t ∪ s_e, cc)
  where s_t = [x ↦ s_e t₀ ∨ (x ∧ s_e (¬t₁))]
        (s_e, cc) = unify₀ (t₀ ∧ t₁) xs
        t₀ = [x ↦ 0] t
        t₁ = [x ↦ 1] t
```

Fig. 4. Boolean unification (unify $t \simeq 0$)

$[u \mapsto \bullet, v \mapsto \bullet]$. (Recall that attributes are boolean expressions.) However, none of these solutions is most general, and it is not obvious that the equation $u \vee v \simeq \bullet$ even has a most general unifier, which means we would lose principal types. Fortunately, unification in a boolean algebra is unitary [9]. In other words, if a boolean equation has a solution, it has a most general solution. In the example, one most general solution is $[u \mapsto u, v \mapsto v \vee \neg u]$.

There are two well-known algorithms for unification in a boolean algebra, known as Löwenheim's formula and successive variable elimination [9,10]. For our core system either algorithm will work, but when arbitrary rank types are introduced and we need to use skolemization [11], only the second is practical.[4] Temporarily using the more common 0 for false (not unique) and 1 for true (unique), to unify two terms p and q it suffices to unify $t = (p \wedge \neg q) \vee (\neg p \wedge q) = 0$. This is implemented by $\texttt{unify}_0$, shown in Fig. 4, which takes a term t in a boolean algebra a and the list of free variables in t as input, and returns a substitution and the "consistency condition", which will be zero if unification succeeded.

6 On Subtyping

In this section we compare our approach to subtyping with that of Clean [2] and to that of our previous paper on the topic [3]. Consider again the function `dup`:

```
dup :: t^× -×-> (t^×, t^×)^u
dup x = (x, x)
```

In Clean `dup` has the same type, but that type is interpreted differently. Clean's type system uses a subtyping relation: a unique type is considered a subtype of a non-unique type. That is, we can pass in something that is unique (such as a unique `Array`) to a function that is expecting a non-unique type (such as `dup`).

The fact that a unique array can become non-unique is an important feature of a uniqueness type system. A non-unique array can no longer be updated,

[4] Löwenheim's formula maps any unifier to a most general unifier, reducing the problem of finding an MGU to finding a specific unifier. For the two-element boolean algebra that is easy, but it is more difficult in the presence of skolem constants. For example, assuming that u_R and v_R are skolem constants, and w is a uniqueness variable, the equation $u_R \vee v_R \simeq w$ has a trivial solution $[w \mapsto u_R \vee v_R]$, but we can no longer guess this solution by instantiating all variables to either true or false.

but can still be read from. However, adding subtyping to a type system leads to considerable additional complexity, especially when considering a contravariant/covariant system with support for algebraic data types (such as Clean's). It becomes simpler when considering an invariant subtyping relation, but we feel that subtyping is not necessary at all.

In our previous paper, we argued that the type of `dup` should be

$$\texttt{dup} \ :: \ t^u \xrightarrow[\times]{u_f} (t^\times, t^\times)^v$$

The (free) uniqueness variable on the t in the domain of the function indicates that we can pass unique or non-unique terms to `dup`. Since it is always possible to use a uniqueness variable in lieu of a non-unique attribute, an explicit subtyping relation is not necessary.

But there is a catch. As we saw in Sect. 4.2, functions with unique elements in their closure must be unique, and must *remain* unique: they should only be applied once. In Clean, this is accomplished by regarding unique functions as *necessarily unique*, and the subtyping is adjusted to deal with this third notion of uniqueness: a necessarily unique type is *not* a subtype of a non-unique type. Hence, we cannot pass functions with unique elements in their closure to `dup`.

Unfortunately, when `dup` gets the type from our previous paper it *can* be used to duplicate functions with unique elements in their closure. Therefore we introduced a second attribute on the function arrow, indicating whether the function had any unique elements in its closure. The typing rule for application enforced that functions with unique elements in their closure (second attribute) were unique (first attribute). That means that functions with unique elements in their closure can be duplicated, but once duplicated can no longer be applied.

This removed the need for subtyping, but that advantage was offset by the additional complexity introduced by the second uniqueness attribute on arrows: the additional attribute made types more difficult to read (especially in the case of higher order functions).

An important contribution of the current paper is the observation that this additional complexity can be avoided if we are careful when assigning types to library functions. For example, a function that returns a new empty array should get the type

$$\texttt{newArray} \ :: \ \texttt{Int} \xrightarrow{\times} \texttt{Array}^u$$

rather than

$$\texttt{newArray} \ :: \ \texttt{Int} \xrightarrow{\times} \texttt{Array}^\bullet$$

Similarly, the function that clears all elements of an array should get the type

$$\texttt{resetArray} \ :: \ \texttt{Array}^\bullet \xrightarrow{\times} \texttt{Array}^u$$

rather than

$$\texttt{resetArray} \ :: \ \texttt{Array}^\bullet \xrightarrow{\times} \texttt{Array}^\bullet$$

An `Array` that is polymorphic in its uniqueness can be passed to `resetArray` as easily as it can be passed to `dup` (of course, a shared array still cannot be

passed to `resetArray`). If we are careful never to *return* a unique array from a function, we will always be able to share arrays. We still do not have an explicit subtyping relation but we get the same functionality: the subtyping is encoded in the type of `Array`, rather than in the type of `dup`.

Not all functions should be so modified. For example, many functions with side effects in Clean have a type such as

$$\texttt{fun} :: \cdots \rightarrow (\texttt{World}^{\bullet} \rightarrow \texttt{World}^{\bullet})$$

where the `World` is a token object representing the world state. It never makes sense to duplicate the world, which can be enforced by returning a unique `World` (rather than a `World` which is polymorphic in its uniqueness).

It may seem that a disadvantage of our approach is that we can no longer take advantage of more advanced sharing analyses. For example, given

$$\begin{aligned} &\texttt{isEmpty} && :: \texttt{Array}^{u} \xrightarrow{\times} \texttt{Bool}^{\times} \\ &\texttt{shrink}, \texttt{grow} && :: \texttt{Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{u} \end{aligned}$$

sharing analysis has been applied correctly to the following definition [2]:

$$\texttt{f arr = if isEmpty arr}^{\otimes} \texttt{ then shrink arr}^{\odot} \texttt{ else grow arr}^{\odot}$$

Even though there are three uses of `arr` within `f`, only one of the two branches of the `if`-statement will be executed. Moreover, the condition is guaranteed to be evaluated before either of the branches, and the shared ($\otimes$) annotation on `arr` means that the array will not be modified when the condition is evaluated.

However, this example uses `arr` at two different types: $\texttt{Array}^{\times}$ within the condition and $\texttt{Array}^{\bullet}$ within both branches. This works in Clean because $\texttt{Array}^{\bullet}$ is a subtype of $\texttt{Array}^{\times}$. In our previous proposal [3], this works because a unique term can always be considered as a non-unique term. In our new proposal however, this program would be rejected (since $\texttt{Array}^{\bullet}$ does not unify with $\texttt{Array}^{\times}$).

However, we can take advantage of the fact that we have embedded our core system in an advanced type system that supports first class polymorphism (Sect. 7). We want to use a polymorphic value ($\texttt{arr} :: \forall u.\texttt{File}^{u}$) at two different types within a function: the classic example of a higher rank type [11]. Our example above typechecks if we provide the following type annotation:

$$\texttt{f} :: \forall v.\, (\forall u.\, \texttt{Array}^{u}) \xrightarrow{\times} \texttt{Array}^{v}$$

The function `f` now *demands* that the array that is passed in is polymorphic in its uniqueness. That is reasonable when we consider that we are using the array at two different types in the body. Moreover, since we regard all unique objects as necessarily unique, it is also reasonable that we cannot pass in a truly unique array to `f`.

Of course there is a trade-off here between simplicity (and ease of understanding) of the type system on the one hand and usability on the other. Since the user must provide a type annotation in order for the definition of `f` to typecheck, the type system has arguably become more difficult to use. However, this case is rare enough that the additional burden on the programmer is small, and a case can be made that it is useful to require a type annotation as it is non-obvious why the function definition is accepted.

7 Implementation in Morrow

We have integrated our type system in *Morrow*, an experimental functional language developed by Daan Leijen.[5] Morrow's type system is HMF [12], which is a Hindley/Milner-like type system that supports first class polymorphism (higher rank types and impredicativity). As such, it is an alternative to both Boxy Types [13] and MLF [14]. However, unlike boxy types, it is presented as a small logical system which makes it easier to understand, and at the same time it is much simpler than MLF. Although HMF is quite a good fit with our type system, we have also experimented with integrating it into other type systems. For example, we we have a prototype implementation of a variant on the type system of this paper that uses the arbitrary rank type system from [11].

As it turns out, the implementation of our type system in Morrow is agreeably straightforward. This provides strong evidence for our claim that adding uniqueness typing to an existing compiler, and more importantly, extending uniqueness typing with advanced features such as higher rank types and impredicativity, poses little difficulty when using the techniques from this paper.

We outline the most important changes we had to make to Morrow:

- We modified the kind checker to do kind inference for our new kind system (mostly a matter of changing the kinds of type constants)
- We implemented sharing analysis, annotating variables with information on how often they are used within their scope (once or more than once)
- We modified the rules for variables and abstraction, so that shared variables must be non-unique, and abstractions become unique when they have unique elements in their closure. To be able to do the latter, all the typing rules had to be adapted to return the *fv* structure from Sect. 4. Variables that are used at a polymorphic uniqueness (a type of the form $\forall u.t^u$ for some t) must be treated as if they were unique for the purposes of *fv*.
- Let bindings had to be adapted to remove the variables bound from *fv*. Moreover, the type of every binding in a recursive binding group must be non-unique (as is standard in a uniqueness type system [2]).
- Most of the work was in modifying the types of the built-in functions and the kinds of the built-in types, and adding the appropriate type constants (such as `Attr`) and kind constants ($\mathcal{T}, \mathcal{U}$). However, all of these changes were local and did not affect the rest of the type checker.
- Unification had be adapted to do boolean unification, as explained in Sect. 5. In addition, it is necessary to simplify boolean expressions, so that for example $t^{u \vee \times}$ is simplified to t^u. This is important because if no simplification is used the boolean expressions can quickly get complicated. Fortunately, we can use an independent module for boolean unification and simplification. When unifying $a \simeq b$, it suffices to check the kinds of a and b, and if they are $\mathcal{U}$, to call the boolean unification module. Therefore, boolean unification does not in any way complicate the unification algorithm of the type checker.

[5] Unfortunately we cannot currently make the source available due to licensing issues.

– Morrow uses System F (with pattern matching) as its typed internal language. Although the "attributes are types" approach of Sect. 3 means that the internal language does not need to change, Morrow also includes a System F type checker to ensure that the various phases of the compiler generate valid code. This type checker had to be adapted in a similar way to the main type checker.

The majority of these changes were local (did not require any significant refactoring of the compiler), and none of the changes were complicated. The fact that we can treat both vanilla types and uniqueness attributes as types (of different kinds) really helped: modifying the kind checker was straightforward, we got the additional expressive power described in Sect. 3 virtually for free, we did not have to introduce an additional universal quantifier for uniqueness attributes (and thus avoided having to modify operations on types such as capture avoiding substitution or pretty-printing), etc.

8 Soundness

To prove soundness, we use a slightly modified (but equivalent) set of typing rules.[6] Rather than giving different typing rules for variables marked as used once or used more than once, we do not mark variables at all but enforce that unique variables are used at most once by splitting the environment into two in rule App. Non-unique variables can still be used more than once because the context splitting operation collapses multiple assumptions about non-unique variables (rule $\textsc{Split}^{\times}$). This presentation of the type system is known as a *substructural* presentation because some of the structural rules (in this case, contraction) do not hold. The presentation style we have used, using a context splitting operation, is based on that given in [15], where it is attributed to [16].

The soundness proof for a type system states that when a program is well-typed it will not "go wrong" when evaluated with respect to a given semantics. We are interested in a lazy semantics; often the call-by-name lambda calculus is used as an approximation to the lazy semantics, but it is not hard to see that we will not be able to prove soundness with respect to the call-by-name semantics. For example, consider

```
(\x. (x, x)) (f y)
```

In the call-by-name semantics, this term evaluates to

```
(f y, f y)
```

But when we allow for side effects, these two terms have a different meaning. In the first, we evaluate $f\,y$ once and then duplicate the result; in the second, we evaluate $f\,y$ twice (and so have the potential side effect of f twice). Accordingly, the types of both terms in a uniqueness type system are also different. In the

[6] The syntax-directed presentation using sharing marks is easier to understand and more suitable for type inference. However, it is not usable for a soundness proof. Such a distinction between a syntax-directed and a logical presentation is not uncommon, and has been used before in the context of uniqueness typing [2].

Term language

$e ::= x \mid \lambda x \cdot e \mid e\, e$	term
$A ::= \lambda x \cdot e \mid \text{let } x = e \text{ in } A$	answer
$E ::= [] \mid E\, e \mid \text{let } x = e \text{ in } E \mid \text{let } x = E_0 \text{ in } E_1[x]$	evaluation context

Syntactic convention

$(\text{let } x = e_1 \text{ in } e_2) \equiv (\lambda x \cdot e_2)\, e_1$

Evaluation rules

$\mapsto$ is the smallest relation that contains VALUE, COMMUTE, ASSOC and is closed under the implication $M \mapsto N$ implies $E[M] \mapsto E[N]$.

(VALUE)	$\text{let } x = \lambda y \cdot e \text{ in } E[x]$	$\mapsto \{(\lambda y \cdot e)/x\}\, E[x]$
(COMMUTE)	$(\text{let } x = e_1 \text{ in } A)\, e_2$	$\mapsto \text{let } x = e_1 \text{ in } A\, e_2$
(ASSOC)	$\text{let } y = (\text{let } x = e \text{ in } A) \text{ in } E[y]$	$\mapsto \text{let } x = e \text{ in let } y = A \text{ in } E[y]$

Substructural typing rules

$$\frac{}{\Gamma, x : t^u \vdash x : t^u|_{x:u}} \quad \textsc{Var}$$

$$\frac{\Gamma, x : a \vdash e : b|_{fv} \qquad fv' = \triangleright_x\, fv}{\Gamma \vdash \lambda x \cdot e : a \xrightarrow{\vee\, fv'} b|_{fv'}} \quad \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : a \xrightarrow{u} b|_{fv_1} \quad \Delta \vdash e_2 : a|_{fv_2}}{\Gamma \circ \Delta \vdash e_1\, e_2 : b|_{fv_1 \circ fv_2}} \quad \textsc{App}$$

Context splitting

$$\frac{}{\emptyset = \emptyset \circ \emptyset} \quad \textsc{Split}^{\emptyset} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : t^{\times} = (\Gamma_1, x : t^{\times}) \circ (\Gamma_2, x : t^{\times})} \quad \textsc{Split}^{\times}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : t^{\bullet} = (\Gamma_1, x : t^{\bullet}) \circ \Gamma_2} \quad \textsc{Split}_1^{\bullet} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : t^{\bullet} = \Gamma_1 \circ (\Gamma_2, x : t^{\bullet})} \quad \textsc{Split}_2^{\bullet}$$

Fig. 5. Call-by-Need Semantics

first, f may or may not be unique, and must have a non-unique result (because the result is duplicated). In the second, f cannot be unique (because it is applied twice) and may or may not return a unique result.

Traditionally [2] a graph rewriting semantics is used to prove soundness, but this complicates equational reasoning. Fortunately, it is possible to give an algebraic semantics for lazy evaluation. Launchbury's *natural semantics for lazy evaluation* [17] is well-known and concise, but is a big-step semantics which makes it less useful for a soundness proof. The call-by-need semantics by Maraist *et al.* [4] is slightly more involved, but is a small-step semantics and fits our needs perfectly. The semantics is shown in Fig. 5.

Unfortunately, due to space limitations we can only give a summary of the proof here. A full formal proof, written using the *Coq* proof assistant, can be found in a separate technical report [18].

Theorem 1 (Progress). *Suppose e is a closed, well-typed term ($\emptyset \vdash e : \tau|_{fv}$ for some τ and fv). Then either e is an answer or there exists some e' such that $e \mapsto e'$.*

Proof. The easiest way to prove progress is to prove a weaker property first: for every term e, e is an answer, there exists some e' such that $e \mapsto e'$, or $e = E[x]$ for some x. This weaker property can be proven by a complete structural induction on e; the proof is laborious but not difficult. To prove progress using the weak progress property, we just need to rule out the last possibility. However, if $e = E[x]$ for some x, and $\emptyset \vdash e : \tau|_{fv}$, then we must have $x \in \emptyset$, which is impossible. □

The proof of preservation is more involved and we can only give a brief outline here. The main lemma that we need to be able to prove preservation is the substitution lemma:

Lemma 1 (Substitution). *If $\Gamma, x : a \xrightarrow{\bigvee fv_2} b \vdash e_1 : \tau|_{fv_1, x:\bigvee fv_2}$, x is free in e_1, and $\Delta \vdash \lambda y \cdot e_2 : a \xrightarrow{\bigvee fv_2} b|_{fv_2}$, then $\Gamma \circ \Delta \vdash \{(\lambda y \cdot e_2)/x\}\, e_1 : \tau|_{fv_1 \circ fv_2}$.*

The proof is by induction on $\Gamma, x : a \xrightarrow{\bigvee fv_2} b \vdash e_1 : \tau|_{fv_1, x:\bigvee fv_2}$ and is not trivial. The essence of the proof is that if $(\lambda x \cdot e_1)(\lambda y \cdot e_2)$ is well-typed, then either x occurs once in e_1, in which case we can substitute $\lambda y \cdot e_2$ for x without difficulty, or x occurs more than once in e_1. In that case, x must have a non-unique type, which means that $\lambda y \cdot e_2$ must be non-unique, and therefore the function cannot have any unique elements in its closure—or equivalently, that e_2 be typed in an environment where every variable has a non-unique type. Since $\Delta = \Delta \circ \Delta$ if all assumptions in Δ are non-unique, this means that we can type the result term even when $\lambda y \cdot e_2$ is duplicated.

Armed with the substitution lemma, we can prove preservation:

Theorem 2 (Preservation). *If $\Gamma \vdash e : \tau|_{fv}$ and $e \mapsto e'$ then $\Gamma \vdash e' : \tau|_{fv}$.*

Proof. By induction on $e \mapsto e'$. The cases for Commute, Assoc, and the three closure rules (one for each of the non-trivial evaluation contexts) are reasonably straightforward. The case for Value relies on the substitution lemma. □

A full formalization of the calculus extended with (let-bound or first-class) polymorphism is future work.

9 Related Work

There is a large body of related work; we can only discuss the most relevant.

There are two recent papers on uniqueness typing: Harrington [19] presents a categorical semantics for a uniqueness type system like Clean's, and Hage *et al.* [20] present a generic type system that can be instantiated to support either sharing analysis or uniqueness typing.

In both systems all unique terms can be coerced to non-unique terms. As observed in Sect. 6 it is possible to allow this, but one must be careful with partially applied functions which may have unique elements in their closure.

In the type system from Hage *et al.*, functions with unique elements in their closure must be unique; however, these functions can then be coerced to be non-unique and can be applied an arbitrary many times; no special provision is made to prohibit this. Thus, it is possible to define a function `dup!` of type

$$\texttt{dup!} :: t^\bullet \to (t^\bullet, t^\bullet)^v$$

```
dup! x = (\f -> (f ⊥, f ⊥)) (const x)
```

The authors suggest that the problem may be remedied by introducing an additional attribute on arrows, like we suggested in our previous paper (see also Sect. 6)—and they adopt this solution in a later paper [21]. It remains to be seen whether a similar solution to the one we propose in the current paper is possible for their system. The central thesis of their paper is a duality between uniqueness typing and sharing analysis, and it is not clear whether this duality is preserved when removing subtyping.

Harrington suggests a different solution to the problem of partial application. Two distinct sorts of functions are introduced: ones that can have unique elements in their closure (of type $a \multimap b$) and ones that cannot (of type $a \Rightarrow b$). Functions of type $a \Rightarrow b$ do not pose any problems and can safely be applied many times (and potentially return unique results).

Functions with unique elements in their closure can also be applied multiple times, but their result must be non-unique if they are applied more than once. While this means that it is no longer possible to define `dup!`, this approach is not sufficient to guarantee referential transparency. For example, consider a function `closeFile` which returns a boolean indicating whether the file was already closed:

$$\texttt{closeFile} :: \texttt{File}^\bullet \xrightarrow{\times} \texttt{Bool}^\times$$

In Harrington's system, the following program would be accepted

```
f file = (\g. g ⊥, g ⊥) (\x. closeFile file)
```

even though it is not referentially transparent (it would be rejected in our type system). It is accepted because the `closeFile` *always* returns a non-unique result, and hence the restriction that functions that are used more than once must return a non-unique result makes no difference (and hence is not enough to guarantee referential transparency). It may be difficult to modify Harrington's system to adopt a solution similar to the one we propose: subtyping between unique and non-unique terms is fundamental to Harrington's formalization.

Uniqueness typing is often compared to linear (or affine) logic [22]. Although both linear logic and uniqueness typing are substructural logics, there are important differences. In linear logic, variables of a non-linear type can be coerced to a linear type (dereliction). Harrington phrases it well: in linear logic, "linear" means "will not be duplicated" whereas in uniqueness typing, "unique" means "has not been duplicated". According to Wadler: "Does this mean that linearity is useless for practical purposes? Not completely. Dereliction means that we

cannot guarantee a priori that a variable of linear type has exactly one pointer to it. But if we know this by other means, then linearity guarantees that the pointer will not be duplicated or discarded" [22, Sect. 3].

However, some systems based on linear logic (such as [23]) are much closer to uniqueness typing than to linear logic, and these systems could benefit equally from the techniques presented in this paper (attributes as types, boolean expressions for attributes).

Finally, Guzmán's Single-Threaded Polymorphic Lambda Calculus [24] has similar goals to uniqueness typing, but is considerably more complicated. Much of this complexity comes from trying to support a "strict let" construct where unique (or "single-threaded") terms can be used multiple times at a non-unique (multiple-threaded) type. A detailed discussion of this problem is beyond the scope of this paper; see for example [25, Sect. 9.4] or [26].

10 Conclusions

By treating uniqueness attributes as types of a special kind $\mathcal{U}$, the presentation and implementation of a uniqueness type system is simplified, and we gain expressiveness in the definition of algebraic datatypes. We can recode attribute inequalities (implications between uniqueness variables) as equalities if we allow for arbitrary boolean expressions as uniqueness attributes. This makes type inference easier (unification cannot deal with inequalities, but *can* deal with equalities between boolean expressions). Finally, no explicit subtyping relation is necessary if we are careful when assigning types to library functions: we require that unique terms must never be shared, and make sure that functions never return unique terms (but rather terms that are polymorphic in their uniqueness).

Together these observations lead to an expressive yet simple uniqueness type system, which is sound with respect to the call-by-need lambda calculus. The system can easily be extended with advanced features such as higher rank types and impredicativity. We have integrated our type system in Morrow, an experimental programming language with an advanced type system. The implementation required only minor changes to the compiler, providing strong evidence for our claim that retrofitting our type system to existing compilers is straightforward.

Acknowledgements. We thank Daan Leijen, Paul Levy and Adam Megacz for various insightful discussions, and Arthur Charguéraud for his generous assistance with the formal proof in *Coq*, which uses the proof engineering technique devised by him and others [27].

References

1. Barendsen, E., Smetsers, S.: Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, University of Nijmegen (December 1993)
2. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Math. Struct. in Computer Science 6, 579–612 (1996)

3. De Vries, E., Plasmeijer, R., Abrahamson, D.: Uniqueness typing redefined. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449. Springer, Heidelberg (2007)
4. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. JFP 8(3), 275–317 (1998)
5. Jones, M.P.: A system of constructor classes: overloading and implicit higher-order polymorphism. In: FPCA 1993, pp. 52–61 (1993)
6. Sheard, T.: Putting Curry-Howard to work. In: Haskell Workshop 2005, pp. 74–85. ACM, New York (2005)
7. Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System F with type equality coercions. In: TLDI 2007, pp. 53–66. ACM Press, New York (2007)
8. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL 1982, pp. 207–212 (1982)
9. Baader, F., Niphow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
10. Brown, F.M.: Boolean Reasoning. Dover Publications, Inc. (2003)
11. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. JFP 17(1), 1–82 (2007)
12. Leijen, D.: HMF: Simple type inference for first-class polymorphism. Technical Report MSR-TR-2007-118, Microsoft Research, Redmond
13. Vytiniotis, D., Weirich, S., Peyton Jones, S.: Boxy types: inference for higher-rank types and impredicativity. In: ICFP 2006, pp. 251–262 (2006)
14. Botlan, D.L., Rémy, D.: ML^F: raising ML to the power of System F. In: ICFP 2003, pp. 27–38 (2003)
15. Walker, D.: Substructural type systems. In: Pierce, B. (ed.) Advanced Topics in Types and Programming Languages. The MIT Press, Cambridge (2005)
16. Cervesato, I., Pfenning, F.: A linear logical framework. Inf. Comput. 179(1), 19–75 (2002)
17. Launchbury, J.: A natural semantics for lazy evaluation. In: POPL 1993, pp. 144–154 (1993)
18. de Vries, E.: Uniqueness typing simplified—technical appendix. Technical Report TCD-CS-2008-19, Trinity College Dublin
19. Harrington, D.: Uniqueness logic. Theor. Comput. Sci. 354(1), 24–41 (2006)
20. Hage, J., Holdermans, S., Middelkoop, A.: A generic usage analysis with subeffect qualifiers. In: ICFP 2007, pp. 235–246. ACM, New York (2007)
21. Hage, J., Holdermans, S.: Heap recycling for lazy languages. In: PEPM 2008, pp. 189–197. ACM, New York (2008)
22. Wadler, P.: Is there a use for linear logic? In: PEPM 1991, pp. 255–273 (1991)
23. Turner, D.N., Wadler, P., Mossin, C.: Once upon a type. In: FPCA 1995, pp. 1–11 (1995)
24. Guzman, J., Hudak, P.: Single-threaded polymorphic lambda calculus. In: Logic in Computer Science 1990, June 1990, pp. 333–343 (1990)
25. Plasmeijer, R., van Eekelen, M.: Clean language report (version 2.1)
26. Odersky, M.: Observers for linear types. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 390–407. Springer, Heidelberg (1992)
27. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. SIGPLAN Not. 43(1), 3–15 (2008)

Tabular Expressions and Total Functional Programming

Baltasar Trancón y Widemann and David Lorge Parnas

Software Quality Research Laboratory (SQRL)
University of Limerick, Ireland
http://www.sqrl.ie

Abstract. Tabular expressions are a multidimensional structured notation for complex mathematical definitions of relations or functions. They have been found useful for documenting imperative programs by stating the function or relation that describes the black-box behaviour of those programs. Tools are needed to increase the practicality of this approach to documentation. In order to create tools to check and evaluate tabular expressions, we have investigated functional programming as an implementation paradigm that reflects the semantics of these mathematical expressions faithfully. We explain why and how the restriction to total functions improves the semantic correspondence substantially, and describe the basic design and capabilities of our total functional programming tools for tabular expressions. We demonstrate the practical advantages of totality by giving examples for the especially easy and effective application of well-known code transformation techniques to total functional programs.

1 Introduction

1.1 Context

Our research group is developing methods of producing practical reference documentation for software products and components. Our document contents are defined by a relational model in which each document is required to be a representation of a specified relation. In effect, we are using mathematical descriptions of relations to provide specifications and descriptions of programs written in conventional programming languages.

Key to making these documents readable is a multidimensional form of expressions, which we call *tabular expressions* or just *tables*. These parse complex expressions into arrays of simpler expressions allowing readers to "look up" the information that they seek without understanding the whole expression.

Tools that check and evaluate these expressions would be very useful when these methods are applied and we are looking for effective and efficient implementations of such tools.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 219–236, 2008.

1.2 This Work

This paper reports on our experiences with applying the functional programming paradigm to the construction of tools for tabular expressions. Functional programming is a natural choice because

1. The tasks of checking and evaluating tabular expressions are typical examples of side-effect-free processing and interpretation of structured data.
2. The formal semantic model of tabular expressions, as presented to some degree in the earlier work [1] and more generically in the forthcoming [2], is given largely in terms of functions.
3. The intended application of these expressions is software documentation using a relational model [3] but for most applications, the relations are sufficiently represented by their characteristic predicate, i.e., a boolean-valued total function.

   Note that this does not include direct support for *existential* problems, such as search queries or relational composition.

Our intent is to give a reference implementation of the formal model that is not only executable, but also mirrors the intended semantics and the model's theoretical properties faithfully. We show that our goals can almost, but not quite, be achieved by using a universal functional language. We explain where conventional functional programming falls short, and propose an alternative. A shorter report on an earlier stage of this work can be found in [4].

1.3 Related Work

This is not the first time that the relation between tabular expressions and functional programming has been noticed or exploited. In [5], Kahl presents an inductive approach to tables of certain regular types that is compositional in table content and semantics at the same time. He provides an implementation of table constructors and inductive interpretation in Haskell, and corresponding formal proofs in the proof system Isabelle. Because of the restricted set of constructors, his theory is compact and elegant.

Our current work is intended to implement the more generic table model of [2], that allows *all* constructs of a certain mathematical base language to be used freely in content and semantics of tables. This paper discusses the requirements of such a generic view, and presents preliminary results from the approach we have taken.

2 Example Tabular Expressions

We shall use a simple tabular expression taken from [6] as the running example for explaining the basic usage of tables and the services we expect from an evaluation tool.

Table 1. Power Conditioning (Specification)

$PwrCnd(Prev : bool; Power, K_{in}, K_{out} : real) : bool \equiv$

$Power \leq K_{out}$	$K_{out} < Power < K_{in}$	$Power \geq K_{in}$
false	*Prev*	*true*

The tabular expression depicted as Table 1 is a small, but real example.[1] It specifies a family of sensor control functions of a nuclear reactor shutdown system. As some of the status monitoring logic is only applicable when the reactor is operating near its maximum output power level, some sensors need to be "conditioned in" (activated) above a certain power level, and "conditioned out" (deactivated) below. To avoid *jitter* (many changes separated by very short intervals), an artificial hysteresis effect is introduced by setting the threshold for conditioning in (K_{in}) slightly higher than that for conditioning out (K_{out}). In between the two, the previous state (*Prev*) is maintained. A graph illustrating some change of power over time is depicted in figure 1. The relevant state transitions and their effects on the output signal are marked.

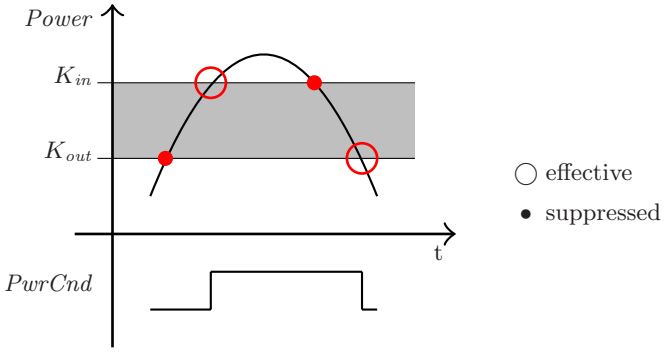


Fig. 1. Power Conditioning (Example Graph)

Table 2 shows a more complex tabular expression. It specifies a software used for testing notebook keyboards in terms of a function. Given a sequence T of keystrokes, the function N yields the next expected keystroke or a test verdict (*pass* or *fail*). The empty sequence is denoted $\_$. The auxiliary functions r and p yield the last keystroke and the subsequence before the last keystroke, respectively. L is the total number of keys to test. *Esc* is the number of the escape key. The two-dimensional organization is crucial for accessing relevant information

[1] Although we have chosen the simplest possible real example to illustrate these expressions, many much more complex tables were used in the inspection of the Darlington Nuclear Power Generation Station described in [6].

Table 2. A Two-Dimensional Function Table with Auxiliary Predicates

$N(T) \equiv$

$T = \_$	$T \neq \_$		
	$N(p(T))=1$	$1<N(p(T))<L$	$N(p(T))=L$

$keyOK(T)$						
$\neg keyOK(T) \wedge$	$\neg keyesc(T) \wedge$	$pkeyOK(T)$		2	$N(p(T))+1$	*Pass*
		$\neg pkeyOK(T) \wedge$ $pkeyesc(T) \wedge$ $ppkeyOK(T)$			$N(p(T))-1$	$N(p(T))-1$
		$\neg pkeyOK(T) \wedge$ $pkeyesc(T) \wedge$ $\neg ppkeyOK(T)$		1	$N(p(T))$	$N(p(T))$
		$\neg pkeyOK(T) \wedge$ $\neg pkeyesc(T)$	1	1	$N(p(T))$	$N(p(T))$
	$keyesc(T) \wedge$	$\neg pkeyesc(T)$			$N(p(T))$	$N(p(T))$
		$pekeyesc(T)$				
		$pkeyesc(T) \wedge$ $\neg pekeyesc(T)$		*Fail*	*Fail*	*Fail*

$$
\begin{aligned}
keyOK(T) &\equiv r(T) = N(p(T)) \\
keyesc(T) &\equiv r(T) = Esc \\
pkeyOK(T) &\equiv keyOK(p(T)) \\
pkeyesc(T) &\equiv keyesc(p(T)) \\
ppkeyOK(T) &\equiv pkeyOK(p(T)) \\
pekeyesc(T) &\equiv N(p(p(T))) = Esc
\end{aligned}
$$

quickly; e.g., the bottom row states that pressing the escape key twice in an error situation, and only doing so, aborts the test.

2.1 Meaning of a Tabular Expression

The concrete syntax for Table 1 in print is deceivingly straightforward; for multidimensional, irregular or simply huge tables, there may not be such an obvious graphical representation.[2] Hence the mathematical table model only represents the abstract syntax of the *content* of the table as an indexed set (aka family or map) of *grids*. Each grid is in turn an indexed set of *cells*, each of which contains a (conventional or nested tabular) expression. A table *type* complements the content to make the tabular expression semantically self-contained. The table type, which may be shared by many similar tables, comprises

[2] The layout shown in Table 2, for example, is the final product of thorough analysis and skillful documentation.

1. an *evaluation term*, i.e., an algorithm for evaluating the table's content, depending on a valuation of free variables,
2. a *restriction predicate*, i.e., a well-formedness condition that a table's content must satisfy for the evaluation algorithm to be applicable.

The table type is an integral part of the table expression. One can consider it as an instance of dynamic typing, or as semantically rigorous meta-data.

The given example table is a one-dimensional instance of the n-dimensional *normal function table* type:

1. It contains two grids of three cells each.
   (a) The upper grid is called a *header* grid that contains predicate expressions.
   (b) The lower grid is called a *main* grid that contains value expression (also of type *bool* in this case).
2. To evaluate the table, choose an index to the header grid, such that
   (a) the selected predicate expression evaluates to *true*,
   (b) then evaluate only the cell of the main grid at the same index.
   For more than one dimension, one index for each header grid would be chosen independently, determining one coordinate of the selected cell of the main grid.
3. The table is well-formed, if
   (a) there is a single main grid and one header grid (per dimension),
   (b) the index set of the main grid is the Cartesian product of the index sets of the header grids (in one dimension, both are equal), and
   (c) each header grid partitions the set of possible variable valuations.

See [1,5,7,8] (in chronological order) for more exact definitions of the normal function table type and other types of tabular expressions.

Figure 2 shows the content of Table 1 as a data term modelling its abstract syntax. The two-layered structure of grids and cells is represented by nested lists of key-value pairs. The index keys 0 for the main grid, 1 for the header grid and 'a', 'b', 'c' for the columns have been chosen arbitrarily. The boxes indicate that some translation of the cell expressions has to take place, see section 5.2. Implementing the table type, as specified above and based on this model, in a functional language is left as an exercise to the reader. See section 5.4 for a suggested interface.

$$\left[\left(0, \left[\left(\text{'a'}, \boxed{\mathit{false}}\right), \left(\text{'b'}, \boxed{\mathit{Prev}}\right), \left(\text{'c'}, \boxed{\mathit{true}}\right)\right]\right), \left(1, \left[\left(\text{'a'}, \boxed{\mathit{Power} \leq K_{out}}\right), \left(\text{'b'}, \boxed{K_{out} < \mathit{Power} < K_{in}}\right), \left(\text{'c'}, \boxed{\mathit{Power} \geq K_{in}}\right)\right]\right)\right]$$

Fig. 2. Abstract Syntax of Table 1

2.2 Tool Requirements

We expect an evaluation tool to enable us to

1. evaluate a tabular expression for a given variable valuation, by applying the evaluation term specified by the table's type to its content,
2. check the restriction predicate, distinguishing two parts for practical reasons:
   (a) clauses that do not depend on variable values but rather on the table shape (called the *static* restriction), to be checked universally for the table's content,
   (b) clauses that do depend on variable values (called the *dynamic* restriction), to be checked specifically for the table's content and a given variable valuation,

all with reasonable efficiency.

We do not expect an evaluation tool to support checking dynamic restrictions universally for all possible variable valuations, thus proving that the table specifies a well-defined function. This is a task for a theorem proving system, and may involve much more complex computations. In [6], based on earlier work [9], the authors show how a flaw in the specified table has been discovered by the automatic theorem prover PVS: the header cells are only a partition of the valuation space, if the (intuitive, but unstated) assertion $K_{out} < K_{in}$ holds. Otherwise, the first and third columns overlap, and the table does not specify a function.

3 The Logic Behind Tabular Expressions

If tabular expressions are to be used for describing real problems, they must be able to deal with partial functions. Partial functions can lead to undefined expressions, and there are many ways to handle undefinedness in logic, e.g., by having three or more truth values.

The meaning of partial functions in tabular expressions here is the one defined in [10]. It was chosen to give the simplest possible expressions in the table cells. It can be summarized as follows:

1. There is a special *undefined* value ($*$), distinct from all proper values of interest. This value is assigned to the application of a partial function to arguments outside its domain.
2. The domain of partial functions never contains $*$. This implies that the result of a partial function is $*$ whenever one of its arguments is $*$, i.e., functions are strict. In other words, a partial function is treated as if it were a *total* function whose range includes $*$.
3. Predicates are treated differently from functions. A *primitive* predicate is simply *false* if any of its arguments is $*$. By λ-abstraction and negation, composite predicates can be built that do not have this property, but are still total Boolean-valued functions. Consequently, the truth value of a formula is always *true* or *false*, but never $*$. I.e., predicates are non-strict.

Note that $(=)$ is considered a primitive predicate, so (by the third rule) the seemingly trivially true predicate expression $f(x) = f(x)$ is *not* true if x is outside the domain of f. On the other hand, the equation $f(x) = y$ is logically equivalent to $F(x, y)$ where F is the characteristic predicate for f. It has been argued in [10] and later work that this interpretation of partial functions is particularly concise and useful for writing software descriptions and specifications in the tabular notation.

This semantic decision has consequences for the construction of an effective universal evaluation algorithm for tabular expressions. The intuitively appealing representation of $*$ by the element $\perp$ of standard domain-theoretic semantics does not work as intended: Since $\perp$ is also assigned to expressions that cannot be evaluated effectively, e.g., a nonterminating recursive function application, writing a program that would evaluate arbitrary predicates in the formalism becomes as hard as solving the halting problem, i.e., impossible without restrictions.

1. The *pragmatic* solution is to use a universal language to implement the model, accepting some semantic deviations. It is impossible to preclude undetermined predicate expressions in this case; so the responsibility is placed on the programmer to find the appropriate termination arguments.
2. The *rigorous* solution is to use a restricted language with the "right" semantics to implement the model. If we have to decide whether an expression evaluates to $*$, it has to be an proper value in a calculus of total functions. The advantage of this approach is that properties of the implementation are closely related to (and not much more complex to prove than) properties of the formal model. The price is that one has to obey the restrictions of the implementation language.

4 Total Functional Programming

In [11], Turner expresses similar, albeit more fundamental concerns regarding the relation of universal functional programming calculi and mathematical functions:

> *The driving idea of functional programming is to make programming more closely related to mathematics. [...] The existing model of functional programming [...] is compromised to a greater extent than is commonly recognized by the presence of partial functions.*

He strives for a language that abolishes partial functions, but retains as much as possible of the notational ease of Miranda or Haskell.

A quite different approach to total functions is taken by total function calculi in the style of Martin-Löf's type theory [12] or Coquand's "Calculus of Constructions" (*CC*) [13]. These are closely connected to higher-order logic (via the Curry-Howard isomorphism), a fact that is exploited in constructive proof systems like Coq.

We have chosen a "middle road", employing a rigorous explicit type system like the latter, but focusing on computation (rather than logic) like Turner.

The result is FCN[3], the design and implementation of a practical high-level intermediate language for pure total functions. Like other total languages, it is characterized by the absence of general recursion: The syntax forbids recursive definitions, and the type system forbids fixpoint operators.

The type system of FCN is the pure System F_ω. It differs from the well-known weaker System F [14] or the even weaker Hindley-Milner system by the presence of type-level functions, and from the stronger CC by the absence of value-dependent types. The detailed properties and relations of these type systems can be found in [15], but are not essential to understanding the following sections.

5 Functional Programming Techniques Applied

Limited space prohibits the detailed description of the FCN language. The following subsections can provide only a brief illustration of how our requirements have been mapped successfully onto features of the purely functional paradigm. The following sections demonstrate that total functional programming is not only semantically adequate for our domain of application, but that its benign properties can be exploited in general, in a number of straightforward and effective ways.

5.1 Partial Functions

The logical rules concerning partial functions and total predicates can be implemented in a completely explicit way using a simple *error monad* [16].

1. For each type A, a *dubious* type $A?$ is defined to contain one additional element:
$$\texttt{type}\ A? = A + \{*\}$$
Readers familiar with Haskell will easily recognize this construction as the *Maybe* functor.
2. A partial function $f : A \nrightarrow B$ is represented as a total function $f' : A \to B?$. Consider another partial function $g : B \nrightarrow C$, totalized as $g' : B \to C?$. The composition $g' \circ f'$ is not type-correct, so a canonical transformation
$$\mathit{bind} : \forall\, B, C.\, (B \to C?) \to (B? \to C?)$$
is inserted.[4] It satisfies the strictness law
$$\mathit{bind}(B)(C)(g')(x) = \begin{cases} * & \text{if } x = * \\ g'(x) & \text{if } x \neq * \end{cases}$$

[3] Functional Core Notation.
[4] Cf. Haskell's `(=<<)`.

such that the composition of total functions $bind(g')\circ f'$ correctly implements the composition of partial functions $g\circ f$. The operation *bind* can be extended to the functorial operation of (?) to deal with total functions:

$$lift : \forall\, B, C.\, (B \to C) \to (B? \to C?)$$

3. For primitive predicates, a different canonical transformation

$$prim : \forall\, A.\, (A \to bool) \to (A? \to bool)$$

is used to compose them with partial functions. It satisfies the non-strictness law

$$prim(A)(p)(x) = \begin{cases} false & \text{if } x = * \\ p(x) & \text{if } x \neq * \end{cases}$$

Apart from reflecting the intended semantics precisely, this approach has several additional benefits:

1. Algebraic simplification laws that do not hold for the original implicit notation are restored. These include the aforementioned tautology $f(x) = f(x)$, as well as general β-reduction.
2. A single boolean-valued function $f : A \to bool$ can be re-used to define both a partial function and a total predicate, by exchanging $lift(A)(f) : A? \to bool?$ and $prim(A)(f) : A? \to bool$.
3. Unlike in the original, untyped first-order approach of [10], there is no ambiguity which symbols are primitive predicates and thus subject to the non-strictness rule: they are explicitly qualified with *prim*.

5.2 Cells and Variables

Tabular expressions are used to define functions and relations, hence they are likely to contain (free) variables. In a language with first-order functions, the grid structure of a table and the functionality of individual cells can be separated cleanly by *closure conversion*, aka *lambda lifting*: The open expression in each cell is turned into a function of the table's variables, which can then be stored in a data structure. In the given example table, the effect is that the phrase

$$\lambda\, Prev : bool; Power, K_{in}, K_{out} : real.\, \cdots$$

is prepended to each cell expression. A variable valuation then takes the form of an argument vector that is passed uniformly to all cells of the table.

To keep expressions simple, we take the liberty to apply another, more invasive transformation, namely a *tuple conversion*. Instead of lifting each free variable to an individual function argument, we provide a single valuation record argument, here invariably called x, and assume the existence of suitable selector functions to extract variable values. By this transformation, e.g., the topmost leftmost cell of the example table becomes the closed unary predicate

$$\lambda\, x : X.\, Power(x) \leq K_{out}(x)$$

where X is a suitable product or record type.

5.3 Data Types and Recursion

It is well-known that algebraic data types can be represented in pure λ-calculus by the Church encoding. However, because of efficiency issues (some access operations slow down from $\mathcal{O}(1)$ to $\mathcal{O}(n)$, where n is the total size of the data structure), they have never been considered seriously for practical purposes.[5] Intermediate languages, but also theoretical calculi are therefore usually extended with primitive data type constructs, e.g., see [18]. Church encoding is also not suitable to be used directly in a strongly typed language based on the Hindley-Milner type system, which is too weak to assign useful types to Church-encoded data. E.g., the following innocuous Haskell function becomes ill-typed, if Haskell's built-in list type is replaced by a Church-encoded one:

$$foo\ l\ =\ l\ +\!\!+\ [sum\ l]$$

Figure 3 shows the full Haskell program. It is not typeable, because a variable in the monomorphic type of argument l is unified with both $List\ \alpha$ and α by its two uses. This weakness is not shared by System F-like type systems.

$$\begin{array}{lcl}
nilC\ f\ e & = & e \\
consC\ h\ t\ f\ e & = & f\ h\ (t\ f\ e) \\
foldC\ f\ e\ l & = & l\ f\ e \\
sumC & = & foldC\ (+)\ 0 \\
appC & = & foldC\ consC \\
singletonC & = & (`consC`\ nilC) \\
foo\ l & = & appC\ l\ (singletonC\ sumC\ l)
\end{array}$$

Fig. 3. Chuch-Encoding of Lists in Haskell

For an intermediate language that is to accommodate not only evaluation, but also symbolic reasoning about functions in a variety of target systems, efficiency is a minor concern compared with the purity and simplicity of the semantic foundation. Hence we consider definitions of the usual data types and their operations in terms of Church-encoded pure functions perfectly acceptable in the context of FCN.

The theoretical advantages of the Church encoding over other representations are

1. that its operations can be typed in a type system of pure functions,
2. that it implements primitive recursion on a data type simultaneously with the construction of data items, and
3. that it is a natural representation for partial evaluation and theorem proving.

As an example, consider the recursive definition of lists of elements of some type A and its constructor operations:

[5] With the outstanding and very recent exception of the SAPL interpreter [17].

$$\begin{aligned} \mathit{List} &= \lambda\, A.\, \forall\, B.\, (A \to B \to B) \to B \to B \\ \mathit{nil} &: \forall\, A.\, \mathit{List}(A) \\ \mathit{cons} &: \forall\, A.\, A \to \mathit{List}(A) \to \mathit{List}(A) \end{aligned}$$

The constructors are implemented by the following functions:[6]

$$\begin{aligned} \mathit{nil} &= \lambda\, A.\, \lambda\, B.\, \lambda\, f.\, \lambda\, e.\, e \\ \mathit{cons} &= \lambda\, A.\, \lambda\, h.\, \lambda\, t.\, \lambda\, B.\, \lambda\, f.\, \lambda\, e.\, f(h)\big(t(B)(f)(e)\big) \end{aligned}$$

with $\mathit{nil}(A)$ denoting the list of zero As, and $\mathit{cons}(A)(h)(t)$ the list of As with first element h and rest list t. That this definition effectively supports primitive recursion can be seen by considering the following function (Haskell experts will recognize it as *foldr* by its type signature)

$$\begin{aligned} \mathit{fold} &: \forall\, A, B.\, (A \to B \to B) \to B \to \mathit{List}(A) \to B \\ \mathit{fold} &= \lambda\, A.\, \lambda\, B.\, \lambda\, f.\, \lambda\, e.\, \lambda\, l.\, l(B)(f)(e) \end{aligned}$$

and it is straightforward to verify that the associated laws hold:

$$\begin{aligned} \mathit{fold}(A)(B)(f)(e)(\mathit{nil}) &= e \\ \mathit{fold}(A)(B)(f)(e)\big(\mathit{cons}(h)(t)\big) &= f(h)\big(\mathit{fold}(A)(B)(f)(e)(t)\big) \end{aligned}$$

Since FCN is a domain-specific language, the question whether primitive recursion provides sufficient computational power should be answered by surveying the application domain. The calculations that make up a typical table evaluation term or restriction predicate deal with finding or counting elements (cells) in a data structure (grid) that have a specific property. These are ideal applications of primitive recursion. None of the table types described in [2] involve any operations for which a primitively recursive form is not easily found. Besides, we doubt that a table type definition involving non-primitively recursive functions would be considered good practice of formal documentation by software engineers, because such a definition would also impede the human reader's understanding.

An FCN program can be transformed in various ways during translation to an executable target language to compensate the inefficiency of the Church-encoding:

1. By replacing all constructors and deconstructors, and thus the effective representation of data items, by primitive implementations available in the target language. The purity of the functional code is lost in translation, if the target language allows nondeterminism or side effects.
2. By replacing the original Church-encoding with a more efficient variant, as in [17]. The resulting code is still purely functional, but not typeable, and may require explicit recursion.

[6] Technically, type checking in System F requires explicit declaration of the types of all λ-bound variables. They have been omitted here to make the example more human-readable.

Some data types, most notably floating point numbers, are supported by virtually any target language, but far too cumbersome to Church-encode. If these are needed, one can give abstract declarations of their access functions (giving types, but no implementation) in FCN, and resolve them to primitive implementations in each supported target language. The disadvantage of this solution is that generic support for evaluation and theorem proving is lost.

5.4 Program Specialization

By virtue of its type system, the FCN language has the *strong normalization property*: All programs have a unique β-normal form, and every sequence of β-reductions reaches this normal form in finitely many steps. As a useful corollary, a FCN program that is run in interpreted or compiled form terminates. This is only part of the story, because we usually do not require functional languages to perform reductions under a λ-abstraction at run-time. But strong normalization implies also that any partial evaluation, that is performed at compile-time and involves such reductions under λ-abstraction, terminates. This allows the transformation of FCN programs using vastly more aggressive reduction strategies than possible in compilers for non-normalizing languages.

λ-abstraction and application are the only primitive constructs of FCN, and β-reduction is the only semantic operation. Hence it subsumes a variety of transformations that appear under different names in code optimization literature, most notably *inlining*, *constant folding*, *constant propagation*, *copy propagation*, *loop unrolling* and *dead code elimination*. A straightforward, brute-force application of *beta*-reduction without any binding-time analysis [19] yields a surprisingly effective partial evaluator that exploits the potential for specialization inherent in tabular expressions to a high degree.

Consider the following record type that represents tabular expressions:

$$Table = \left\{ \begin{array}{ll} content & : Content \\ statrest & : Content \to bool \\ dynrest & : Content \to X \to bool \\ eval & : Content \to X \to R? \end{array} \right\}$$

The first component is the table content, i.e., a two-level data structure containing cells. The remaining three components comprise the table type. The type parameter X is the type of variable valuations, and R is the range type of the function described by this table. Since the table type components are given as functions of arbitrary content, there seems to be little potential for specialization. But since we are only interested in one point of each of these functions, we can find a more specific representation:

$$Table' = \left\{ \begin{array}{ll} content & : Content \\ statrest & : bool \\ dynrest & : X \to bool \\ eval & : X \to R? \end{array} \right\}$$

A simple mapping applies the universal table type components to the specific content and creates starting points for partial evaluation, leaving only the variable valuation as run-time input.

$$compile : Table \to Table'$$

$$compile(T) = \left\{ \begin{array}{ll} content & = content(T) \\ statrest & = statrest(T)\big(content(T)\big) \\ dynrest & = dynrest(T)\big(content(T)\big) \\ eval & = eval(T)\big(content(T)\big) \end{array} \right\}$$

Several auxiliary phases have been added to complement partial evaluation in an optimizing whole-program compiler.

1. The access operations of several simple and ubiquitous data types are exempt from inlining to allow for the replacement of their representation, as discussed in section 5.3.
2. Since these operations are no longer handled by β-reduction, a programmable term rewriter has been added. The specification of elimination rules (e.g., the previously given rules for *fold*) for non-inlined operations yields a controlled form of δ-reduction.

   Note that in the general, non-total case, rewriting applications of strict functions is problematic and usually requires a termination proof for the arguments. In the domain of total functions, there is no such problem.
3. A final phase of *common subexpression elimination*, or β-expansion, is applied to clear up the inevitable duplication of code.

Applying the whole-program compiler to the example table results in the inlining of 147 auxiliary functions, 788 β-reductions and 1586 rewrite rule applications. A specialized program P equivalent to the following pseudo-code is obtained, minor technical details aside.

1. The static restriction, which states that the index set of the main grid is the product of index sets of the header grids, is evaluated completely.

$$statrest(P) = true$$

2. The dynamic restriction, which states that each header grid partitions the space of variable valuations, is specialized to a check that the three header cells (A, B, C) cover all cases and are pairwise disjoint.

$$\begin{array}{l} dynrest(P) = \lambda\, x. \\ \qquad \begin{array}{ll} \text{let} & A = Power(x) \le K_{out}(x) \\ & B = K_{out}(x) < Power(x) < K_{in}(x) \\ & C = Power(x) \ge K_{in}(x) \\ \text{in} & (A \vee B \vee C) \wedge \neg\big((A \wedge B) \vee (A \wedge C) \vee (B \wedge C)\big) \end{array} \end{array}$$

3. The evaluation term is specialized to the computation of the selected index I in the header grid, and the selection of the corresponding cell F of the main grid.

$$
\begin{aligned}
eval(P) = \lambda\, x.\ & \\
& \text{let } A = Power(x) \leq K_{out}(x) \\
& \quad\ B = K_{out}(x) < Power(x) < K_{in}(x) \\
& \quad\ C = Power(x) \geq K_{in}(x) \\
& \quad\ I\ = \text{if} \quad\ A \text{ then } 0 \\
& \qquad\qquad \text{else if } B \text{ then } 1 \\
& \qquad\qquad \text{else if } C \text{ then } 2 \\
& \qquad\qquad \text{else} \qquad\qquad\ * \\
& \quad\ F = \text{if} \quad\ I = 0 \text{ then } \lambda\, x.\, true \\
& \qquad\qquad \text{else if } I = 1 \text{ then } \lambda\, x.\, Prev(x) \\
& \qquad\qquad \text{else if } I = 2 \text{ then } \lambda\, x.\, false \\
& \qquad\qquad \text{else} \qquad\qquad\qquad \lambda\, x.\, * \\
& \text{in } F(x)
\end{aligned}
$$

The separation of these two steps seems slightly redundant in one dimension, but is quite natural in the general, multi-dimensional case.

This code is significantly simpler that the pair of abstract syntax and table type implementation it has been derived from.

Note that all computations that need to iterate over a data structure by a primitively recursive operation are evaluated at compile-time. It can be shown that this is a general property of the normal function table type, and also of some other common table types described in [2]. This eliminates the need for loops or recursion in the evaluation, and for induction in symbolic reasoning.

6 Tool Support

6.1 Programming System

Programming in FCN is supported by tools, most notably a parser, type checker, interpreter and compiler. All of these are implemented in Java. The compiler produces Java code that runs on the JVM, together with a small runtime library. Figure 4 shows a compiled version of the example table. It is controlled by a

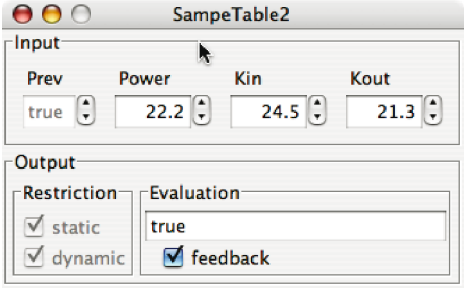


Fig. 4. Power Conditioning (Simulation Screenshot)

GUI that is derived directly from the function signature, and will be generated automatically by a future version of the tools.

We have considered the possibility of compiling FCN to another high-level functional language to leverage the efficiency of mature compilers and run-time systems. We have chosen not to do so, because

1. the mapping to a type system that is weaker with respect to polymorphism, e.g. that of standard Haskell, is difficult to automate,
2. interoperation of the generated code with the other existing tools for tabular expressions (all implemented in Java) would become difficult, and
3. we have not found efficiency to be problematic, especially compared with symbolic reasoning tools.

A library of about 1000 lines of FCN code defines the ubiquitous basic types and operations: booleans, natural and integer numbers, tuples, lists, monads, etc., all in terms of Church encoding.

A second level of library code, about 500 lines of FCN, defines the table model in terms of standard functional data structures and operations, as well as some common table types, including the multi-dimensional normal function table type used in the example, and reusable (polymorphic, higher-order) components for such table types. This library will be extended in the future to support other table types.

A tabular expression is simply data structured according to the model, containing functions at the cell level. Evaluation and restriction checking are completely generic operations, because all semantic information is explicit in the "type" part of the table data.

Tabular expressions that describe software behaviour can be "animated" with compiled FCN code to produce simulations, test oracles or prototypes.

6.2 Documentation as Input

The FCN language, with its focus on higher-order functions, its strongly formal type system and absence of "syntactic sugar", is designed deliberately as an intermediate representation facilitating semantically safe interchange of tabular expressions. It is not, however, intended to be used as an input format for the software engineering practitioner.

The design of our table tool collection is organized around a central documentation repository or server, called the *kernel*, that provides peripheral input, output and analysis tools with access to tabular expressions via an API or exchange file format.

The FCN programming system currently supports the import of tabular expressions in a file format based on OpenMath [20], a common XML-encoded abstract syntax notation for mathematical expressions. The import translation deals automatically with several concerns that have to be addressed explicitly at the FCN level:

1. Free variables in table cells are detected and λ-lifting is performed.
2. Computations with partial functions are lifted to the monadic level, inserting *bind*, *lift* and *prim* where appropriate.

3. Simple patterns of recursive definition are recognized and replaced by applications of a primitive recursion operator. This is still work in progress.

6.3 Total Functions and Theorem Proving

There is ongoing work [21] to represent the formal model of tabular expressions as a theory in the proving system PVS. This would complement the services of the evaluation tools by allowing to prove properties of tables universally for a class of variable valuations.

Because of the close similarity between the calculi of total functions and the higher-order logic of PVS, and because of the explicit treatment of partiality issues in the FCN implementation, large parts of the table model's design carry over to PVS directly. The FCN type checker has proved a valuable tool for quick consistency checking in the design process, helping to keep consistency proof obligations in the PVS theory tractable. Furthermore, the specialization of a table by partial evaluation

1. compensates the many layers of indirection and generic auxiliary functions in the table model library and
2. unrolls recursive operations that would otherwise require inductive reasoning,

thus greatly enhancing the scalability of symbolic reasoning about tables.

7 Conclusion

The work described in this paper is an experimental use of functional programming in the creation of software engineering tools. The approach has provided a formulation of the mathematical model of tabular expressions that can reflect semantics precisely, but is also directly and effectively executable. The strict type system has proven a valuable consistency check. The features of functional programming that are supposed to support abstraction and reuse in functional programming, namely parametric polymorphism and higher-order functions, have found essential use, e.g., as primitive recursion operators and monadic liftings.

We have also found the notion of total functional programming, that is looked upon with some scepticism by most of the community, to be quite feasible for this specific application. The absence of general recursion does not impede the construction or interpretation of tables unduly. The pervasive use of recursion operators even encourages a point-free programming style.[7]

The absence of infinite reduction sequences greatly simplifies the implementation of both run-time and compile-time evaluation strategies. A safe and effective technique for improving run-time control flow by means of *laziness analysis* will be described in a forthcoming paper. The implementation of a partial evaluator for exhaustive compile-time program simplification has been extremely

[7] An obvious benefit from the viewpoint of the functional programmer, but of questionable merit for the software engineer.

straightforward, compared to the vast amount of sophisticated partial evaluation strategies and heuristics for Turing-complete languages found in literature.

Finally, we have found that an underlying calculus of total functions greatly reduces the impedance mismatch between the implementation of a formalism and its formalization in a proof system, making it attractive in general for projects that involve both evaluation and verification.

Acknowledgments

Thanks to Dennis Peters, Mark Lawford and other SQRL members for helpful discussions. Thanks also to members of the special interest group on programming languages of the *Deutsche Gesellschaft für Informatik*, who provided valuable feedback on a preliminary presentation of the subject.

References

1. Parnas, D.L.: Tabular representation of relations. CRL Report 260, McMaster University (1992)
2. Balaban, A., Bane, D., Jin, Y., Parnas, D.L.: Mathematical model of tabular expressions. SQRL draft (2007) (to be published), `David.Parnas@ul.ie`
3. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Trans. Softw. Eng. 20(12), 948–976 (1994)
4. Trancón y Widemann, B., Parnas, D.L.: Tabular expressions and total functional programming. In: Hanus, M., Brassel, B. (eds.) 24. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, Institut für Informatik, Christian-Albrechts-Universität Kiel (2007)
5. Kahl, W.: Compositional syntax and semantics of tables. SQRL Report 15, McMaster University (2003)
6. Lawford, M., Froebel, P., Moum, G.: Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Formal Methods in System Design (accepted for publication, October 2004), `http://www.cas.mcmaster.ca/~awford/papers/`
7. Zucker, J.: Transformations of normal and inverted function tables. Formal Aspects of Programming 8, 679–705 (1996)
8. Janicki, R., Wassyng, A.: On tabular expressions. In: CASCON 2003: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, pp. 92–106. IBM Press (2003)
9. Jing, M.: A table checking tool. SERG Report 384, McMaster University (2000)
10. Parnas, D.L.: Predicate logic for software engineering. IEEE Trans. Softw. Eng. 19(9), 856–862 (1993)
11. Turner, D.A.: Total functional programming. Universal Computer Science 10(7), 751–768 (2004)
12. Martin-Löf, P.: A theory of types. Technical Report 71–3, University of Stockholm (1971)
13. Coquand, T., Huet, G.: The calculus of constructions. Technical Report RR-0530, INRIA Rocquencourt (1986)

14. Girard, J.Y.: Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In: Fenstad, J.E. (ed.) Proceedings of the Second Scandinavian Logic Symposium, North-Holland, Amsterdam (1971)
15. Barendregt, H.: Lambda calculi with types. In: Abramsky, Gabbay, Maibaum (eds.) Handbook of Logic in Computer Science, Clarendon, vol. 2 (1992)
16. Spivey, M.: A functional theory of exceptions. Sci. Comput. Program 14(1), 25–42 (1990)
17. Jansen, J.M., Koopman, P., Plasmeijer, R.: Efficient interpretation by transforming data types and patterns to functions. In: Proceedings of Trends in Functional Programming (TFP) (2006)
18. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Transactions on Programming Languages and Systems 21(3), 528–569 (1999)
19. Mogensen, T.: Binding Time Aspects of Partial Evaluation. PhD thesis, University of Copenhagen (1989)
20. Buswell, S., Caprotti, O., Carlisle, D.P., Dewar, M.C., Gaëtano, M., Kohlhase, M.: The OpenMath Standard (Version 2.0). The OpenMath Society (2004)
21. Peters, D.K., Lawford, M., Trancón y Widemann, B.: An IDE for software development using tabular expressions. In: Proceedings of CASCON (2007)

Speculative Inlining of Predefined Procedures in an R5RS Scheme to C Compiler

Marc Feeley

Dépt. d'informatique et de r.o., Université de Montréal, Canada

Abstract. The semantics of some dynamic programming languages, including Python, JavaScript, and R5RS Scheme, make it hard for a compiler to inline predefined procedures without compromising the semantics of the language. In the case of Scheme, many existing compilers can only achieve good execution speed by assuming that variables bound to predefined procedures are never mutated. This paper presents a *speculative inlining* approach which is portable and achieves good performance while fully conforming to the semantics of Scheme. It has been implemented in a mature Scheme to C compiler and we report on its performance on a large benchmark suite, both in execution speed and code size.

1 Introduction

Functional abstraction is useful for designing modular programs but the procedure call mechanism which implements the abstraction barrier has a run time cost for setting up parameters, directing the control flow to and from the procedure, and returning any results to the caller. The compiler can reduce the cost by *inlining* the procedure at the call site in the caller. This eliminates the need for the call/return control flow instructions, and it uncovers additional opportunities for optimization because the copy of the procedure body placed at the call site can be specialized for the actual parameters of that call.

We distinguish two kinds of inlinable procedures: *predefined procedures* provided by the language (e.g. `sqrt` and `map`), and *user procedures* whose definition must be given explicitly in the program. This classification covers language support operations, such as arithmetic, I/O, memory allocation and method dispatch, by treating them as inlinable predefined procedures. This paper addresses the problem of inlining predefined procedures in the R5RS Scheme language [11].

Some aspects of Scheme make inlining predefined procedures tricky. According to the semantics of Scheme the evaluation of `(+ x y)` decomposes into these steps: get the values of the variables `+`, `x`, and `y`, respectively t_1, t_2, and t_3, then check that t_1 is a procedure, and then call t_1 with the parameters t_2 and t_3. This usually adds `x` and `y` because the global variable `+` is initially bound to the addition procedure. During program execution it is possible to bind the variable `+` to a non-procedure value or to a different procedure, for example `(set! + list)`. After this assignment, the expression `(+ x y)` will in fact call the predefined procedure `list` and thus construct a two element list. This form

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 237–253, 2008.

of late binding may be surprising, but it is sometimes useful as explained in Section 2. This problem is not specific to Scheme; indeed Python [14], JavaScript [2] and other scripting languages implement this form of late binding.

Predefined global variables may be mutated at run time in any part of the program, at the read-eval-print loop, in modules loaded dynamically using the predefined `load` procedure, and in S-expressions constructed and evaluated at run time by the predefined `eval` procedure. To achieve a better static analysis of programs, a Scheme compiler could adopt a static linking model by forbidding dynamic loading and `eval`, and not offering an interactive read-eval-print loop. This would allow a whole program analysis to determine conservatively that a given predefined variable is never mutated. Although this simplifies procedure inlining, it reduces the system's flexibility and it deviates from the Scheme semantics.

A popular alternative approach is the use of command-line flags and non-standard program annotations to force the compiler to assume the predefined variables contain their initial bindings. For example, the `(standard-bindings)` annotation of Gambit-C [3], the `--prim` flag of PLT Scheme's compiler [6], and the "benchmark mode" of Scheme48 [10]. This assumption is so common that it is the default compilation mode of CHICKEN [15], and the only compilation mode of Bigloo [13] which are both Scheme to C compilers.

Another aspect of Scheme which hinders the inlining of predefined procedures is the generic nature of fundamental operations such as `+` and `equal?`. Scheme supports a rich set of numerical types (infinite precision integers, rational, real, and complex) and the notion of exactness. Consequently most predefined procedures for performing arithmetic operations have non-trivial definitions which dispatch on the type and exactness of its arguments, type check the arguments, check for overflows, raise exceptions when appropriate, perform memory allocation, and so on. For space reasons it is unreasonable to fully inline arithmetic procedures. This is problematic because many programs need to do simple arithmetic on small exact integers for counting or indexing vectors.

To alleviate this problem many systems extend Scheme with a *fixnum* numerical type, which is a fixed-width exact integer type typically a few bits less than the natural word size of the machine, and procedures to operate on fixnums, such as `fx+` to add two fixnums, `fx<` to compare two fixnums, etc. Fixnum operations usually have a simple definition and they are fast because they do not require boxing and unboxing, and they have few special cases (overflow checking is typically the only special case). Some systems also provide a *flonum* numerical type, which is a fixed-precision inexact real type typically represented as a boxed machine floating point number. It is reasonable to inline fixnum and flonum operations because they take roughly the same space as a general procedure call.

The handling of fixnums and flonums varies considerably between systems, most notably in the width of fixnums and the handling of overflows (some systems detect fixnum overflows and signal an error, while others silently wraparound). For this reason it is difficult to write portable and fast programs even across systems that support fixnums and flonums. Moreover, fixnums cannot be used in applications where the computations result in bignum integers that exceed

the fixnum range even though most of the time the computations are within the fixnum range (financial calculations, number theoretic algorithms, etc).

A final concern in Scheme to C compilers, as considered here, is the high cost for implementing procedure calls. In order to correctly implement tail-calls the C code generated cannot in general directly translate Scheme calls into C calls. This is due to the fact that the global call graph of the program is partially known when a module is compiled (because of separate compilation, dynamic loading of modules, `eval` and mutation). Moreover, when compiling to C, some compilation techniques such as runtime code generation cannot be used.

To address these problems we have designed a speculative inlining algorithm which fully obeys the semantics of R5RS Scheme and does not rely on any program annotations, although it can take advantage of annotations to further improve performance. This algorithm has been integrated into the Gambit-C Scheme to C compiler. With no annotations, some programs approach the speed of hand tuned code with annotations and fixnum/flonum specific operations.

This paper describes the speculative inlining algorithm, how it integrates into the Gambit-C Scheme compiler and its performance. Section 2 explains situations where mutation of predefined global variables is useful. Section 3 discusses aspects of Gambit-C's compiler which interact with the speculative inlining algorithm which is described in detail in Section 4. Finally Section 5 reports on the performance both in execution speed and code space.

2 Mutation of Predefined Global Variables

The ability to mutate predefined global variables is consistent with Scheme's minimalistic philosophy. Using a single namespace for procedures and values (a "Lisp$_1$" [7]) is conceptually simpler than using two namespaces (a "Lisp$_2$"). Disallowing mutation of predefined global variables would increase the language's complexity by adding a special class of global variables. Moreover, mutation of predefined global variables is useful, as shown in the following examples.

Debugging – Scheme programs are often structured as a set of procedures defined at top-level. These procedures are bound to global variables and each call references the appropriate variable to get the procedure to call. Tracing calls to these procedures can be done by setting the variable to a new procedure which calls the old one and also displays the arguments and result of the call. This can be achieved by defining and using a `trace` macro as shown in Figure 1 (a).

Defining new types – There are no constructs in R5RS Scheme to define new types. Portable programs must represent new types using a predefined type, usually vectors. New types defined this way are not distinct because they cannot be distinguished from other new types and the vector used for their representation. A common solution is to use a unique tag at the head of the vector to identify the type unambiguously from other new types. Moreover, the `vector?` type predicate must be redefined to distinguish plain vectors from vectors representing the new types. Figure 1 (b) shows how a 2D point type can be defined.

```
> (define-syntax trace
    (syntax-rules ()
      ((trace var)
       (set! var (wrap 'var var)))))
> (define (wrap var proc)
    (lambda args
      (let ((r (apply proc args)))
        (write (cons var args))
        (display " ==> ")
        (write r)
        (newline)
        r)))
> (define (f n) (* (+ n 1) (+ n 2)))
> (trace f)
> (trace +)
> (f 10)
(+ 10 1) ==> 11
(+ 10 2) ==> 12
(f 10) ==> 132
132
```

(a) Debugging

```
(define old-vector? vector?)

(define (instance? obj tag)
  (and (old-vector? obj)
       (>= (vector-length obj) 1)
       (eq? (vector-ref obj 0) tag)))

(define pt (list 'pt)) ; unique tag
(define (make-pt x y) (vector pt x y))
(define (pt? obj) (instance? obj pt))
(define (pt-x p) (vector-ref p 1))
(define (pt-y p) (vector-ref p 2))

(set! vector?
      (lambda (obj)
        (and (old-vector? obj)
             (not (pt? obj)))))

(pt? (make-pt 11 22)) ⇒ #t
(vector? (make-pt 11 22)) ⇒ #f
```

(b) Defining new types

Fig. 1. Predefined global variable mutation examples

Overloading – Overloading of predefined procedures can be achieved easily with mutation. For example, `append` can be extended to allow concatenation of strings by setting the `append` variable to a procedure which either calls the `append` or `string-append` procedures depending on the type of the arguments.

Some scripting languages based on interpreters also support the redefinition of primitive functions. Figure 2 shows simple examples for the JavaScript and Python languages. In both cases the variable `abs`, whose initial binding is the function computing the absolute value, is modified to contain a different function.

```
Math.abs = Math.sqrt
alert(Math.abs(25)) // prints "5"
```

(a) JavaScript

```
abs = hex
print abs(25) # prints "0x19"
```

(b) Python

Fig. 2. JavaScript and Python examples

3 The Gambit-C System

3.1 System Architecture

The Gambit-C system [3] is a Scheme to C compiler based implementation of R5RS Scheme designed to be very portable while achieving good execution speed when programs are compiled.

A large part of the runtime system (roughly 50 kLOC), including the interpreter, debugger, bignum library, and all predefined procedures, is written in Scheme. The compiler is also written in Scheme (roughly 25 kLOC). The rest of

the system (roughly 40 kLOC), including the garbage collector, operating system interface, and foreign function interface is written in portable C.

Many macros which abstract away from the specifics of the platform are defined in the `gambit.h` header file: the use of the `gcc` C compiler and its extensions, the machine's natural word size and endianness, the width of each numeric type, the definition of virtual machine instructions, the representation of objects, etc. This header file plays a key role in the compilation process. The C files produced by the Gambit-C compiler are entirely composed of calls to macros defined in `gambit.h`. This allows a very late binding of the behavior of the generated code. Indeed, a C file produced by the Gambit-C compiler on a given machine does not have to be changed when it is compiled on a machine with a different C compiler, a different operating system, or different endianness and word size (for practical reasons the word size is currently limited to 32 and 64 bits). Porting to an unconventional C compiler typically only requires small changes to `gambit.h`.

Gambit-C supports separate compilation. In particular the runtime system's Scheme code is contained in 9 modules which are separately compiled. The modules of the runtime system and of the user can be statically linked to form an executable program. A running program can also load user modules dynamically and possibly more than once, to simplify debugging. Moreover, because the interpreter is written in Scheme [4], interpreted code and compiled code can freely call each other without compromising the Scheme semantics.

To implement tail-calls and continuations, Scheme calls cannot be translated directly into C calls. The runtime system manages a stack of Scheme continuation frames explicitly and independently from the C stack. The C code generated by the compiler is partitionned into a number of *host C procedures*. Depending on system build options, there is either a single host C procedure per Scheme module (*single host* mode) or one host C procedure per top-level Scheme procedure in the module (*multiple host* mode). Each host procedure contains a number of control points, which can either be procedure entry points or continuation return points. Trampolines are used to allow arbitrary jumps to a destination control point without C stack growth. Host procedures are only called from a dedicated *dispatcher* procedure. To jump to control point P, the current host returns to the dispatcher which then calls the host procedure containing P (i.e. the depth of the C call chain is never more than two). Upon entry to the host, a `switch` statement or a computed `goto`, if `gcc` is used, jumps to P in the host. There are a few optimizations to this basic approach which exploit locality, i.e. when P is in the same host. Regardless of the compilation mode and optimizations, calls to predefined procedures in the runtime are expensive because they necessarily require a non-local jump from the user program. The high cost of calling predefined procedures makes speculative inlining particularly attractive.

3.2 Extensions to Scheme

Gambit-C supports several extensions to R5RS Scheme. Some of the notable extensions are preemptive multithreading and a foreign function interface.

Low-Level Procedures. Many low-level procedures meant primarily for the implementation of the runtime system are provided. These procedures typically perform a simple operation and are unsafe because they do not validate their arguments. For this reason, they are given an easily recognized name that is outside the standard Scheme identifier syntax (i.e. they cannot be found in an R5RS conformant program). These low-level procedures have names that begin with two hash signs. Here are a few examples:

- `##fx+` is the procedure which performs addition of fixnums. It does not check that the arguments are fixnums and whether there is a fixnum overflow.
- `##fx+?` is the procedure which performs addition of fixnums and checks for fixnum overflow. False (`#f`) is returned on overflow, otherwise the sum (a fixnum) is returned. It does not check that the arguments are fixnums.
- `##car` is a procedure which extracts a pair's `car` field. It does not check that the argument is a pair.
- `##cons` is the low-level procedure constructing pairs.

The duplication that occurs for `##cons` (which is identical to `cons`) and other low-level procedures is motivated by the need to easily distinguish internal low-level procedures from the procedures normally accessed by the user. This is convenient for the binding annotations explained in the next section.

User Annotations. User annotations allow the programmer to force the compiler to assume certain properties about the code. This is useful when the programmer has knowledge that can help the compiler optimize the program. Annotations are specified inside the `declare` form. It is the programmer's responsibility to ensure that these annotations are correct; the compiler does not verify them. The `declare` form can appear anywhere a definition can appear. A `declare` at top-level has a lexical scope that extends until the end of the file. For a local `declare`, the lexical scope extends to the end of the enclosing binding form. Here is a typical use of user annotations:

```
(declare
  (standard-bindings) (extended-bindings) (block) (fixnum) (not safe))
(define z 0)
(define (iota n) (if (= n z) '() (##cons n (iota (- n 1)))))
```

The `(standard-bindings)` annotation asserts that a reference to a global variable predefined in R5RS will result in the corresponding predefined procedure. In other words in `iota` the calls to `=` and `-` will call the R5RS predefined procedures with those names. The `(extended-bindings)` annotation is similar but applies to Gambit-C extended procedures, such as `##fx+`, `##cons`, etc.

The `(block)` annotation asserts that all global variables defined in this file are only mutated in this file. Any global variable defined in a file and not mutated in that file can thus be treated like a constant. This enables constant propagation of global variables (e.g. replacing `z` in `iota` with 0), jump destination determination and inlining of user procedures defined at top-level (e.g. determining that the call to `iota` is a self-recursion).

The `(fixnum)` annotation asserts that all arguments and results of arithmetic procedures are fixnums. The `(not safe)` annotation tells the compiler that it is acceptable to generate unsafe code that could crash the program if some type checks fail. These two annotations in conjunction with the `(standard-bindings)` annotation allow the compiler to replace in `iota` the calls to `=` and `-` with unsafe calls to the fixnum specific procedures `##fx=` and `##fx-` respectively.

With carefully chosen annotations, programs can be made to run very fast, but at the price of safety. This is unacceptable in many situations. Moreover, annotations are brittle and are high maintenance. A small change in the program or in the dataset may invalidate the current set of annotations, but it is tedious and error-prone for the programmer to determine which ones.

In designing the speculative inlining algorithm, our goal was to improve the speed of execution without requiring that the programmer resort to annotations that are unsafe or otherwise change the semantics of the language (which includes all annotations described in this section).

3.3 Compiler Architecture

The compiler follows a conventional architecture. The source code is parsed and macros are expanded to produce an abstract-syntax tree (AST), which is transformed and annotated by subsequent passes. The AST is then traversed to generate the code for the Gambit Virtual Machine [5]. Low-level optimizations are then performed on this intermediate representation (dead code and common code elimination, instruction reordering, jump cascade removal, etc.) and finally it is expanded into C code in the form of calls to macros defined in `gambit.h`. The AST after all transformations is optionally pretty-printed as an S-expression.

The AST can represent expressions with no source code equivalent. Specifically, there is an AST node type representing procedure constants. These nodes are generated to refer to the procedure objects that exist at run time. Both predefined procedures and user procedures (but not closures) can be denoted. For example, the AST corresponding to this source code:

```
(let () (declare (standard-bindings)) (cons 11 22))
```

is transformed into an AST representing a call to a procedure constant denoting the predefined procedure `cons` (i.e. there is no longer a reference to the global variable `cons`). We use a box to denote procedure constants in the new AST:

```
(let () (declare (standard-bindings)) (cons 11 22)) → ('cons 11 22)
```

Note that $X \rightarrow Y$ will be used to mean "AST X is transformed into AST Y", where X and Y are external representations of ASTs possibly containing procedure constants. The different passes which transform the AST are briefly explained below. They are executed in the order of presentation.

Assignment Conversion. This pass introduces cells for local variables (including parameters) that are mutated. An assignment to a local variable is replaced with a mutation of the corresponding cell. This simplifies the implementation of

closures and continuations which share mutated local variables. The remaining passes can assume that local variables are never mutated.

Beta Reduction. This pass performs simple beta reductions of the code. The following transformations are done.

- **Constant and copy propagation:** When it is known that a variable V is never mutated and V is bound to X which is either a constant or a variable that is never mutated, references to V are replaced with X. For example:

  `(let ((x 5)) (let ((y x)) (+ y y)))` → `(+ 5 5)`
  `(let () (declare (standard-bindings)) +)` → `'+`

- **Constant folding:** When a constant predefined procedure is called, and all the arguments are constants of the correct type, and the procedure does not have side-effects, the call is replaced by a constant equal to the compile-time application of the procedure to the arguments. For example:

  `('+ 5 5)` → `10`

  There are subtle semantic issues which hinder constant folding. Calls to predefined procedures which allocate their result (e.g. `list` and `append`) are not constant folded because this would not preserve the uniqueness of the result (in the sense of `eq?`). Specifically, in Scheme:

  `(eq? (list 5) (list 5))` ≠ `(eq? '(5) '(5))`

  Because the target platform is not known at the time of the Scheme compilation, constant folding is tricky for procedures whose meaning is dependent on the target platform. This is specifically a problem for the fixnum operations because the width of a fixnum depends on the target machine's word size (30 bit fixnums on 32 bit machines, and 62 bit fixnums on 64 bit machines). An exact integer that does not fit in a 30 bit fixnum and that fits in a 62 bit fixnum is a bignum on 32 bit machines and a fixnum on 64 bit machines. So the `##fixnum?` procedure, which tests if its argument is a fixnum, is only constant folded when its argument is small enough to fit in a 30 bit fixnum or larger than fits in a 62 bit fixnum:

  `('##fixnum? 123)` → `#t`
  `('##fixnum? 1000000000000)` *is not constant folded*
  `('##fixnum? 2305843009213693952)` → `#f`

  Constant folding is also performed on conditional expressions, that is the `if`, `and`, and `or` special forms. For example:

  `(if (and #f (f 2)) 123 (g 3))` → `(g 3)`

- **Inlining of user procedures:** When it is known that a given variable V is never mutated and V is bound to a lambda-expression, calls to V are replaced by calls to a copy of the lambda-expression. As an additional condition, the size of the new call (measured in number of nodes in the AST)

must not be larger that a certain factor F of the size of the original call in the source code. By default F is 3, but the programmer can modify this with the `(inlining-limit` F`)` user annotation. For example:

```
(let ((f (lambda (x) (+ x x)))) (f 5)) → (+ 5 5)
```

To improve the effectiveness of these beta reductions, processing generally starts at the leaves of the AST and progresses towards the root. For binding forms, the bound values are processed before the body. Finally, the top-level procedures are processed in the reverse order of their dependencies. If procedure `f` contains a call to procedure `g`, which contains a call to procedure `h`, then `h` is processed first, then `g`, and then `f`.

Lambda Lifting. This pass transforms local user procedures using the lambda lifting transformation [9]. This eliminates the creation of closures for lambda-expressions bound to local variables when these local variables are only referenced in the operator position of calls. The lambda-expressions are modified so that they take their free variables as explicit parameters. All calls to these variables are also modified to pass the value of the free variables.

4 Speculative Inlining

Speculative inlining of predefined procedures is performed as an AST transformation pass just before assignment conversion.

4.1 Basic Approach

Our approach capitalizes on the high likelihood that predefined global variables contain the corresponding predefined procedure. When a predefined procedure is speculatively inlined, the inlined code must be guarded by a *run time binding test* to verify that the variable is indeed bound to the expected predefined procedure.

If the binding test fails, the inlined code is not appropriate and a normal procedure call using the global variable must be performed. For correct handling of tail-calls, this call must be a tail-call with respect to the original call.

If the binding test succeeds, the inlined code is executed. In the ideal case this code will perform the work required of the predefined procedure and return the required result. It is possible however that the inlined code encounters an exceptional case, such as an argument of the wrong type, or a complex case that would be too space inefficient to handle inline (such as a fixnum arithmetic operation overflowing into the bignum range). We will call these conditions the *inlining conditions* of the procedure. When the inlining conditions do not hold the execution can fall back to a normal procedure call. We require that the inlined code only perform side-effects after verifying the inlining conditions. As an example, here is the speculative inlining of `car`:

```
(f (car (g 5))) → (f (let ((x (g 5)))
                       (if (and ('##eq? car 'car)
                                ('##pair? x))
                           ('##car x)
                           (car x))))
```

Falling back to a normal procedure call is not only correct, it ensures that the behavior of a call to a predefined procedure is the same, except for execution speed, whether the procedure is inlined or not: the same exceptions are raised, the same continuation is used, etc. The inlining is purely a compiler optimization that is transparent to the programmer.

4.2 Inlining Scheme's Numeric Procedures

The inlining of Scheme's numeric procedures is problematic because most numeric operations are generic, they can accept several numeric types, and can accept mixed types. In Gambit-C, there are five representations for numbers: exact integers are represented with fixnums and bignums, exact rationals are represented as pairs of exact integers, inexact reals are represented as flonums (64 bit IEEE 754 floating point number), and complex numbers are represented as pairs of reals. Except for fixnums, these representations are memory allocated.

If we take addition as an example, the algorithm for adding two numbers depends on the representation of the numbers to add. It is necessary to dispatch on the type of both arguments to determine how to proceed. In the Gambit-C runtime all 25 cases are laid out to avoid needless representation conversions.

It is unreasonable to inline this much code routinely. Instead, the most likely case must be handled inline, with the less likely cases handled by the fall back. But what constitutes a likely case depends on the nature of the computation. There is a large class of algorithms which process small exact integers (e.g. counting and indexing vectors). On the other hand, scientific applications usually perform the bulk of their computations with inexact reals. The other numeric types (exact rationals and complex numbers) are less useful to handle inline in Gambit-C because the algorithms operating on them are complex and often require procedure calls (e.g. computing the GCD for normalizing rational results).

Consequently, there are two cases that are interesting to handle inline: when all the arguments are fixnums, and when all the arguments are flonums. A set of 5 user annotations is provided to allow the programmer to specify which case is most likely, and which cases to inline:

- `(mostly-fixnum)`: The fixnum case is more likely and is inlined.
- `(mostly-flonum)`: The flonum case is more likely and is inlined.
- `(mostly-fixnum-flonum)`: The fixnum case is more likely than the flonum case, but both are likely and are inlined. The fixnum case is checked first.
- `(mostly-flonum-fixnum)`: The flonum case is more likely than the fixnum case, but both are likely and are inlined. The flonum case is checked first.
- `(mostly-generic)`: The numeric procedures are not inlined.

These annotations are purely advisory. They affect the performance of the code but do not compromise the Scheme semantics. The following example shows the speculative inlining of `+` when the user annotation `(mostly-fixnum-flonum)` is in effect:

```
(let () (declare (mostly-fixnum-flonum)) (f (+ (g 2) (h 3))))
→
(f (let ((x (g 2)) (y (h 3)))
     (if ('##eq? + '+)
         (if (and ('##fixnum? y) ('##fixnum? x))
             (or ('##fx+? x y) (+ x y))
             (if (and ('##flonum? y) ('##flonum? x))
                 ('##fl+ x y)
                 (+ x y)))
         (+ x y))))
```

In the resulting AST, the `##fx+?` predefined procedure is used to perform the fixnum addition and overflow check. If the global variable `+` does not have its standard binding, or a fixnum overflow is detected, or the arguments are not both fixnum or both flonums, then a normal call to `+` is performed. The common code elimination optimization of the compiler will generate compact code by merging all three calls to `+`.

4.3 Inlining Recursive Procedures

The following recursive predefined procedures on lists are speculatively inlined: `assq`, `memq`, `map`, and `for-each`. Both `assq` and `memq` are worth inlining because many Scheme programs rely on them, their definitions are short, and they do not need to call procedures that are not easily inlined (`assv`, `assoc`, `memv`, and `member` are not inlined because they call `eqv?` and `equal?` whose definitions are considerably more complex than `eq?` which is called by `assq` and `memq`).

To avoid too much code expansion, the higher-order procedures `map` and `for-each` are only inlined when they are passed two arguments: the procedure argument and list. They are worth inlining not only because many Scheme programs rely on them but because the inlined code exposes optimization opportunities at the call to the procedure argument which can often avoid an expensive general call. If the procedure argument is a user procedure in the same file then a direct jump to the procedure can be performed (and without a parameter count if it does not take a rest parameter). The procedure argument is also a candidate for inlining, whether it is a user procedure or predefined procedure.

4.4 Interaction with Beta Reduction Pass

Implementing speculative inlining as an AST transformation has the advantage that subsequent transformations can further optimize the inlined code. In particular, the beta reduction pass may simplify the inlined code through constant propagation and constant folding. Consider a slight variation on the previous

example, where the second argument to `+` is the constant 1. The speculative inlining of `+`, followed by constant propagation will give:

```
(let () (declare (mostly-fixnum-flonum)) (f (+ (g 2) 1)))
→
(f (let ((x (g 2)))
     (if ('##eq? + '+)
         (if (and ('##fixnum? 1) ('##fixnum? x))
             (or ('##fx+? x 1) (+ x 1))
             (if (and ('##flonum? 1) ('##flonum? x))
                 ('##fl+ x 1)
                 (+ x 1)))
         (+ x 1))))
```

The calls `('##fixnum? 1)` and `('##flonum? 1)` will then be constant folded to `#t` and `#f` respectively, allowing both `and`s and the `if` guarding the flonum case to be constant folded:

```
(f (let ((x (g 2)))
     (if ('##eq? + '+)
         (if ('##fixnum? x)
             (or ('##fx+? x 1) (+ x 1))
             (+ x 1))
         (+ x 1))))
```

The constant propagation transformation can also make use of user annotations to further improve the code. If we add the annotation `(standard-bindings)` to the previous example, the code at the end of the AST transformations will be:

```
(f (let ((x (g 2)))
     (if ('##fixnum? x)
         (or ('##fx+? x 1) (+ x 1))
         (+ x 1))))
```

5 Evaluation

5.1 Experimental Setup

To evaluate the effectiveness of the speculative inlining approach, we compiled several Scheme benchmarks using the Gambit-C compiler with various user annotations. We are interested in measuring the impact of our approach on the execution speed and also on the code size. We used Gambit-C version 4.2.3 built with the configure options `--enable-single-host` `--enable-char-size=1` and, for comparison, Bigloo version 3.0d built with the configure option `--benchmark=yes`. All tests were performed on a Linux workstation (2 GHz Dual Core AMD Opteron with 16 GB SDRAM) and `gcc` 4.0.2 was used.

Version 4.2.3 of Gambit-C supports the speculative inlining of 101 R5RS predefined procedures, and 128 Gambit-C specific predefined procedures. The

speculative inlining is performed on all the type predicates (e.g. `pair?`, `null?`), most of the R5RS numeric procedures (e.g. `+`, `-`, `*`, `/`, `=`, `<`, `quotient`, `max`, `sqrt`), all the case-sensitive character comparison procedures (e.g. `char=?`), pair, string and vector constructor, accessor and mutator procedures (e.g. `cons`, `cadr`, `string-length`, `vector-set!`), miscellaneous procedures (e.g. `not`, `values`, `eq?`), and the recursive procedures `assq`, `memq`, `map`, and `for-each`.

The benchmarks contain programs representative of typical Scheme applications. There are 41 benchmarks in all. The largest are: `scheme` (Scheme interpreter in Scheme, 1 kLOC), `slatex` (Scheme to LaTeX formatter, 2.3 kLOC), `nucleic` (scientific application [8], 3.5 kLOC) and `compiler` (Scheme compiler, 11.7 kLOC).

To determine the size of the code generated by the Gambit-C compiler, we measured the size of the machine code produced by the C compiler and substracted the code size for an empty program. We did not measure the size of the program's data because it could not easily be isolated from the data of the Gambit-C runtime. The data size should not vary much between settings.

The benchmarks were also run with Bigloo to compare the execution time with a high-performance Scheme compiler. We used the Bigloo compiler options `-O6 -copt -O3 -copt -fomit-frame-pointer` and no explicit type information was used in the source code. This mode gives a semantics that is close to R5RS, but does not fully conform to it because it assumes that none of the predefined global variables are mutated and it does not check for arithmetic overflow.

For both Gambit-C and Bigloo, we set the same initial heap size when executing the compiled program (10 MB), and strings were represented using one byte per character.

Given our goal of achieving the best execution speed without compromising the Scheme semantics, one set of trials with Gambit-C avoided the annotation `(standard-bindings)`, but we tried each of the numeric user annotations: `(mostly-fixnum)`, `(mostly-flonum)`, etc. The `(mostly-fixnum-flonum)` case is used as the baseline because it corresponds to the default when the programmer does not supply any user annotations. Another set of trials was done with those user annotations combined with `(standard-bindings)`, in violation of the R5RS Scheme semantics. This is useful to evaluate the cost of the run time binding test.

We also tried the benchmarks with the set of user annotations that achieve the best speed which we call unsafe mode. That is the `(not safe)`, `(block)`, and `(standard-bindings)` user annotations were used, in addition to benchmark specific annotations for arithmetic (either `(fixnum)` or `(flonum)` as appropriate for the benchmark).

In addition, a trial was done with the speculative inlining transformation disabled. This situation approximates the Gambit-C compiler before the speculative inlining transformation was added. This trial and those using speculative inlining are the only ones which do not violate the Scheme semantics.

In all trials using Gambit-C, the inlining of user procedures was disabled. As a result the programs run slower than they would normally. This is particularly

noticeable for the unsafe mode where loop unrolling and data structure accessor inlining can boost performance relative to the baseline by 13% on average. This was done to avoid side effects of the user procedure inliner which would add noise to the code size measurements. Bigloo performs automatic inlining of primitives and user procedures.

5.2 Experimental Results

The results are given in Table 1. For each combination of benchmark and compilation mode, the execution time and code size are given and the code size is underlined. Lower values are better. To simplify comparison, all measurements are relative to the baseline (i.e. speculative inlining with `(mostly-fixnum-flonum)` but without `(standard-bindings)`). A value of 1 means the same time or space as the baseline. For lack of space in the table the columns for the baseline are omitted since they contain 1 everywhere for time and space. Moreover we omit the columns for `(mostly-flonum-fixnum)` because the time and space were within a few percent of the columns for `(mostly-fixnum-flonum)`. The table is ordered by increasing speedup of the baseline mode compared to the speculative inlining disabled mode.

By examining the speculative inlining disabled column we see that the benchmarks always execute faster with speculative inlining than without. The geometric mean speedup is 6.14, but in several cases the speedup is more than 10, and over 20 for `sum`. The code size is on average 79% larger when speculative inlining is used, and up to 4.5 times the size for `fft`. Overall we view these results positively since among the compilation modes which do not violate the R5RS semantics, speculative inlining is consistently faster while the code size is typically not unreasonably large.

If we now compare the `(mostly-fixnum)` and `(mostly-flonum)` modes with speculative inlining we see that the execution time for the `(mostly-fixnum)` case is better in general but worse on benchmarks which are floating point intensive (`nucleic`, `ray`, `fibfp`, `mbrot`, `sumfp` and `pnpoly`). The ratio can be up to 10 times in favor of `(mostly-flonum)` for `sumfp` and up to 21 times in favor of `(mostly-fixnum)` for `sum` (the same computation as `sumfp` but performed using small integers). In terms of code size the `(mostly-flonum)` case is normally better, by about 5% on average. This is probably due to the absence of an overflow check when operating on flonums.

Interestingly, `fft`, which uses a mix of operations on fixnums and flonums, is about the same speed with `(mostly-fixnum)` and `(mostly-flonum)`. This is explained by the fact that there is an equal number of fixnum and flonum operations, so the same number of non-local jumps result whether the fixnum case or the flonum case is inlined. The execution speed improves by a factor of over 5 when `(mostly-fixnum-flonum)` or `(mostly-flonum-fixnum)` are used. Considering all benchmarks these compilation modes give the best execution speed; 1.34 times faster than with `(mostly-fixnum)` on average and 2.19 times faster than with `(mostly-flonum)` on average. The `(mostly-fixnum-flonum)` mode gives marginally better performance which is why Gambit-C uses it as

Table 1. Relative execution time and relative code size (underlined) for various compilation modes; baseline is speculative inlining and `(mostly-fixnum-flonum)`

	R5RS semantics obeyed						**R5RS semantics violated**							
	Speculative inlining						`(standard-bindings)`						Unsafe	Big-
Program	disabled		+ fix		+ flo		+ fix		+ flo		+ fix-flo		mode	loo
`dynamic`	1.54	.62	1.00	1.00	1.05	1.00	.96	.63	1.02	.63	.96	.63	.81	.29	.25
`slatex`	1.54	.61	.93	.96	.88	.90	.80	.60	.83	.56	1.01	.61	.74	.20	.45
`ctak`	1.75	.91	1.00	.97	1.55	.98	.90	.79	1.45	.75	.89	.81	.83	.48	55.03
`fibc`	2.23	.84	.99	.96	2.26	.95	.96	.80	2.24	.80	.96	.85	.71	.35	13.78
`conform`	2.78	.73	.98	1.00	.96	1.02	.87	.64	.89	.70	.87	.64	.60	.34	.21
`compiler`	2.99	.69	.99	.94	1.47	.90	.87	.67	1.34	.65	.87	.70	.52	.24	.29
`scheme`	3.77	.74	1.00	1.00	1.25	.97	.95	.71	1.16	.68	.92	.71	.69	.27	.43
`nucleic`	3.84	.54	2.48	.81	1.00	.82	2.34	.61	.87	.63	.90	.73	.34	.17	.87
`ray`	3.96	.47	3.32	.70	.99	.74	3.08	.49	.94	.58	.95	.71	.44	.21	1.24
`maze`	4.55	.55	1.02	.84	2.57	.70	.92	.53	2.49	.49	.92	.59	.34	.16	.40
`paraffins`	4.97	.38	1.00	.73	1.04	.72	.96	.44	.99	.42	.95	.53	1.00	.12	1.06
`deriv`	5.14	.54	1.00	1.00	1.16	1.04	.80	.56	.93	.61	.80	.56	.67	.26	1.23
`perm9`	5.34	.57	.99	.92	3.85	.83	.95	.53	3.73	.47	.93	.60	.85	.16	.80
`matrix`	5.36	.58	1.01	.90	2.39	.86	.86	.54	2.18	.53	.86	.62	.71	.28	.72
`dderiv`	5.48	.58	.98	1.00	1.13	1.03	.81	.58	.95	.60	.82	.58	.61	.28	.82
`fibfp`	5.70	.57	5.69	.61	1.02	.98	5.53	.49	1.00	.79	.93	.84	.64	.17	2.78
`lattice`	5.92	.69	.98	.99	.99	1.00	.83	.64	.84	.64	.82	.65	.71	.35	.32
`graphs`	6.17	.57	1.03	.91	2.31	.86	.91	.63	2.22	.61	.94	.66	.67	.23	.94
`earley`	6.20	.47	1.03	.90	2.89	.78	.89	.53	2.77	.43	.96	.55	.78	.17	.72
`peval`	6.43	.57	1.00	.96	1.03	.94	.76	.50	.81	.50	.76	.51	.49	.23	.31
`divrec`	6.44	.72	1.00	1.00	1.06	.89	.87	.64	.91	.55	.87	.64	.52	.16	.85
`simplex`	6.78	.36	1.99	.71	3.52	.55	1.92	.45	3.40	.35	.91	.52	.44	.11	.61
`primes`	6.94	.66	1.01	.99	3.18	.91	.89	.66	3.10	.55	.89	.68	.75	.16	1.11
`cpstak`	6.95	.81	1.01	.96	5.29	.92	.92	.74	5.18	.72	.93	.77	.84	.36	2.71
`browse`	7.16	.61	.97	.99	1.01	.98	.82	.50	.86	.46	.83	.51	.68	.19	.37
`fib`	7.37	.56	.93	.91	7.29	.69	.86	.65	7.03	.52	.93	.76	.38	.13	.62
`tak`	7.51	.67	1.00	.96	5.68	.86	.88	.65	5.70	.59	.89	.69	.34	.18	.50
`mazefun`	8.25	.66	.97	.98	5.70	.92	.89	.63	5.40	.59	.92	.66	.51	.24	.77
`nboyer`	8.30	.57	.98	.98	1.26	.94	.72	.42	.98	.41	.72	.42	.64	.21	.48
`mbrot`	9.18	.31	7.82	.55	2.37	.49	7.63	.39	2.28	.33	.95	.55	.57	.12	3.15
`takl`	9.30	.77	1.00	1.00	1.01	.94	.65	.66	.60	.59	.65	.66	.23	.17	.13
`triangl`	9.68	.48	.97	.88	6.69	.68	.93	.60	6.50	.42	.93	.60	.32	.14	.53
`sumfp`	9.70	.60	9.81	.55	1.03	.90	9.21	.45	.97	.78	.87	.89	.80	.10	4.75
`diviter`	9.75	.70	1.00	1.00	1.06	.89	.74	.62	.78	.53	.75	.62	.68	.16	1.09
`sboyer`	10.46	.57	.98	.98	1.30	.95	.67	.42	.96	.41	.66	.42	.57	.23	.35
`destruc`	11.16	.47	1.06	.85	6.84	.69	.86	.41	6.62	.34	.87	.41	.67	.12	1.05
`fft`	12.32	.22	5.14	.65	5.11	.46	5.04	.47	4.83	.33	.89	.61	.28	.05	2.03
`pnpoly`	12.85	.39	6.84	.70	3.80	.55	6.75	.50	3.56	.37	.89	.61	.23	.08	4.78
`puzzle`	13.29	.40	.98	.73	8.69	.65	.78	.52	8.25	.47	.83	.57	.35	.10	1.71
`nqueens`	14.12	.40	.96	.64	7.01	.65	.76	.33	6.54	.34	.76	.40	.52	.08	.83
`sum`	20.63	.64	.96	.84	21.05	.70	.75	.63	20.25	.51	.71	.79	.29	.07	1.68
geom. mean	6.14	.56	1.34	.86	2.19	.82	1.17	.56	1.99	.53	.87	.62	.54	.18	.94

the default compilation mode. The code size is consistently bigger than with `(mostly-fixnum)` and `(mostly-flonum)`, but by only 16-22% on average.

The cost of the run time binding tests can be evaluated by looking at the column for the `(standard-bindings)` + `(mostly-fixnum-flonum)` compilation mode. This mode generally yields code that is faster than the baseline, by about 13% on average. This mode also generally yields more compact code than the baseline, 38% smaller on average. Our view is that this is an acceptable cost for the run time binding tests which are required for conformance to the Scheme semantics.

The unsafe mode column indicates that with hand tuned user annotations and unsafe code, programs can run considerably faster, by a factor of about 2 on average but in some cases 4 times faster than the baseline (and even 8 times faster when user procedure inlining is enabled). Moreover the code is over 5 times more compact because there remains very few procedure calls in the code (and consequently fewer return points, continuation frame allocations and setup, stack overflow checks, etc. which all contribute to the total code). This shows in our view that speculative inlining does not completely eliminate the need for unsafe user annotations when very high performance and compact code are required. However, speculative inlining does contribute to lessen the urgency to resort to user annotations and promote a more maintainable coding style.

Finally, we can see that the performance of Gambit-C with speculative inlining is reasonably good in absolute terms. It is about 6% slower than Bigloo on average and about 21% slower when we ignore the `call/cc` intensive benchmarks `ctak` and `fibc`.

6 Related Work and Conclusion

Inlining has been used in other dynamically-typed programming languages to improve performance. Most notable is the compiler for SELF [1], an object-oriented dynamically-typed programming language, which uses *message inlining* to speed up message sends by reducing the frequency of method lookups. On the first execution of the messsage send, a normal method lookup is performed to find the correct method to call based on the type of the receiving object. The message send is then *backpatched* to jump directly to this method and the type of the receiving object is saved. On subsequent message sends the method will be called if the type of the new receiving object is the same, otherwise the system reverts to a new method lookup and backpatch. *Selective recompilation* of the program is used when method definitions are changed. All of this requires a complex system architecture, the presence of the compiler in the runtime system, and runtime code generation. More recently the Java HotSpot VM [12] has used a similar inlining-with-recompilation approach and a complex runtime architecture.

Our approach is in comparison much simpler and can be applied in situations where runtime code generation is not an option such as in compilers which generate C code, in memory constrained systems, and embedded systems where the code must be stored in read-only memory. With an extensive experimental evaluation using a mature Scheme system, we have shown that our approach can

be used to correctly implement the R5RS semantics while achieving execution speeds comparable to other Scheme compilers which attain high-performance by violating the R5RS semantics.

Acknowledgements

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

1. Chambers, C.: The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. PhD thesis, Stanford (1992)
2. ECMA. Ecma-262: Ecmascript language specification (1999)
3. Feeley, M.: Gambit-C version 4.2.3. (2008), `http://gambit.iro.umontreal.ca`
4. Feeley, M., Lapalme, G.: Using closures for code generation. Computer Languages 12(1), 47–66 (1987)
5. Feeley, M., Miller, J.S.: A Parallel Virtual Machine for Efficient Scheme Compilation. In: Proc. of the ACM Symposium on LISP and Functional Programming, pp. 119–130 (1990)
6. Flatt, M.: PLT MzScheme: Language manual. Technical Report PLT-TR05-1-v300 (2005), `http://www.plt-scheme.org/techreports/`
7. Gabriel, R.P., Pitman, K.M.: Technical issues of separation in function cells and value cells. Lisp and Symbolic Computation 1(1), 81–101 (1988)
8. Hartel, P.H., Feeley, M., Alt, M., Augustsson, L., Baumann, P., Beemster, M., Chailloux, E., Flood, C.H., Grieskamp, W., Van Groningen, J.H.G., Hammond, K., Hausman, B., Ivory, M.Y., Jones, R.E., Kamperman, J., Lee, P., Leroy, X., Lins, R.D., Loosemore, S., Röjemo, N., Serrano, M., Talpin, J.-P., Thackray, J., Thomas, S., Walters, P., Weis, P., Wentworth, P.: Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. Journal of Functional Programming 6(4), 621–655 (1996)
9. Johnsson, T.: Lambda lifting: transforming programs to recursive equations. In: Proc. of the conference on Functional Programming and Computer Architecture, New York, NY, USA, pp. 190–203 (1985)
10. Kelsey, R., Rees, J.: The Incomplete Scheme 48 Reference Manual (1999)
11. Kelsey, R., Clinger, W., Rees, J.: Revised^5 report on the algorithmic language Scheme. ACM SIGPLAN Notices 33(9), 26–76 (1998)
12. Paleczny, M., Vick, C.A., Click, C.: The Java HotSpot Server Compiler. In: Java Virtual Machine Research and Technology Symposium, USENIX (2001)
13. Serrano, M., Weis, P.: Bigloo: a portable and optimizing compiler for strict functional languages. In: Proc. of the Static Analysis Symposium, pp. 366–381 (1995)
14. Van Rossum, G.: The Python Language Reference Manual, September 2003. Network Theory Ltd. (2003)
15. Winkelmann, F.L.: CHICKEN - A practical and portable Scheme system (2008)

From Contracts Towards Dependent Types: Proofs by Partial Evaluation

Stephan Herhut[1], Sven-Bodo Scholz[1], Robert Bernecky[2], Clemens Grelck[1,3], and Kai Trojahner[3]

[1] University of Hertfordshire, U.K.
{s.a.herhut,s.scholz,c.grelck}@herts.ac.uk
[2] University of Toronto, Canada
bernecky@acm.org
[3] University of Lübeck, Germany
{grelck,trojahner}@isp.uni-luebeck.de

Abstract. The specification and resolution of non-trivial domain constraints has become a well-recognised measure for improving the stability of large software systems. In this paper we propose an approach based on partial evaluation which tries to prove such constraints statically as far as possible and inserts efficient dynamic checks otherwise.

1 Introduction

Resolving domain constraints for operations on arrays is known to be a challenging task. The central challenge is that one of the most frequently used operations, array selection, has value constraints which are, in general, undecidable. In the context of array languages, such as APL [1], J [2] or SaC [3], which support generic operations on n-dimensional arrays, the challenge is even greater, because these languages treat the rank and shape of an array, at least conceptually, as part of the array value.

In APL, conformance checks are purely dynamic. This design decision has a considerable run-time impact, as noted in [4]. In order to avoid the overhead due to dynamic checks, several other approaches have been developed that try to resolve these requirements statically. However, the undecidable nature of the problem forces these approaches to restrict the expressiveness of the language in one way or the other. Some approaches are based on restricted forms of dependent types, such as the *indexed types* proposed by Zenger in [5] or the type system of DML [6]. Other approaches rely on a strict separation of arrays and indices and force all indices to be defined in a rather restricted manner only. This enables languages such as Zpl [7] or Chapel [8] to in many cases avoid run-time checks.

In this paper, we propose a hybrid approach. Rather than restricting either the language or the compiler to programs whose constraints can be statically resolved, we make the compiler resolve and eliminate as many constraints as possible and check the unresolved ones at run-time. For many straightforward programs this yields the same static safety as do strongly typed systems.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 254–273, 2008.

Dynamic checks remain only for those computations that rely on more complex index calculations.

The central idea of our approach is to use partial evaluation as a constraint resolution mechanism. In a first step, all domain constraints are explicitly inserted into the program. At that stage, programs are very similar to programs that contain contracts, as first proposed in the context of EIFFEL [9,10]. In fact, our proposed approach facilitates a seamless integration of arbitrary contracts, as found in several modern languages from the object-oriented domain, *e.g.*, JAVA [11,12] and PYTHON [13].

Subsequently, partial evaluation is applied, with the intent of safely eliminating as many dynamic checks as possible. A detailed analysis of remaining checks allows the programmer to decide if the level of static guarantees is sufficient for the application given. If not, further partial evaluation can be applied, or the program can be re-written in a way so that static resolution becomes feasible. As a nice side-effect, those checks that remain until run-time have been minimised with respect to the actual checks being performed.

We demonstrate this approach in the context of the functional array language SAC. A prototype implementation is included in the current beta release of the SAC compiler[1]. Since the existing compiler for SAC already supports powerful mechanisms for partial evaluation as part of its type system and as part of its optimisation cycle, this implementation required only moderate effort.

The main contributions of this paper are:

- a partial-evaluation-based approach towards static domain guarantees,
- a discussion of the implications of some of the design alternatives for a practical implementation of the proposed approach,
- an outline of a formal transformation scheme for the core language SAC_λ that introduces explicit domain constraints in a contract-like style, and
- an outline of a formal proof of the semantic soundness of this transformation.

The paper is structured as follows. Section 2 identifies some of the challenges of the proposed approach. Section 3 gives a brief introduction to SAC_λ, a stripped-down functional array programming language which has similar syntax and semantics to SAC but is better suited for formal reasoning. Using SAC_λ, Section 4 explores the design space of representing constraints explicitly by contracts. Section 5 discusses different means to insert contracts into the code. A formal presentation of the chosen approach is given in Section 6. Section 7 gives a brief discussion of how partial evaluation is used to resolve contracts. Related work is discussed in Section 8 before Section 9 concludes.

2 Challenges of the Contract Approach

Although the approach to use explicit contracts and to eliminate these by means of partial evaluation seems to be rather straightforward it turns out that a practical implementation of it poses several challenges which need to be addressed.

[1] The compiler is available for download at `http://www.sac-home.org/`

Our implementation as part of the SaC project (`http://www.sac-home.org/`) revealed the following challenges.

Feedback of the verification process. As laid out in the introduction, one of the primary motivations of this work is to obtain static guarantees about the good behaviour of a program. Due to the hybrid nature of the proposed approach, any residual program may be left with unresolved conformity checks. If we still want the programmer to benefit from successfully inferred guarantees, it is essential to provide the programmer with feedback which distinguishes those parts of the program that could be checked statically from those where errors may still occur at run-time. While the identification of potentially unsafe program regions comes almost for free in approaches based on tailor-made inferences, in the proposed approach this requirement poses a challenge. Since we start out from a "blind" insertion of contracts which are, hopefully, optimised away later, we need to make sure that remaining run-time checks can still be related to the original program, even after program optimisation.

Efficient checking at run-time. As pointed out in [4], the elimination of redundant run-time checks can have a vast impact on the overall run-time behaviour of generic array programs. Therefore, we need to make sure that the amount of checking that happens at run-time is reduced as much as possible. For example, a program that contains an element-wise addition of two arrays `A` and `B` and an element-wise subtraction of these should not check more than once that their shapes are identical. Apart from such reuses of entire constraints, we also expect the system to partially evaluate constraints and minimise the actual checking required. One example for such a situation is the selection operation: in generic array programming, selections require that the length of the index vector matches the rank of the array to be selected from and that each component of the index vector is in the proper range of indices for the corresponding array axis. While the former usually can be ensured statically, the latter sometimes has to be postponed until run-time. In those cases, we expect only the value checks to remain in the optimised program.

Stepwise improvements for separately compiled code. Static verification of contracts is often rendered impossible if separate compilation is required. Being based on partial evaluation, we expect our approach to be well-suited for separate compilation without a loss of checking efficiency. Rather than starting out from scratch, it should be possible to take a pre-compiled library version of any program and to further eliminate potentially remaining run-time checks whenever enough information of the calling context becomes available.

Constraint unaware optimisation. One of the main problems of high-level program optimisation is that most optimisations need to carefully observe all domain constraints involved. If these cannot be statically proved, a conservative approach must be taken; this often inhibits application of such optimisations. In the intended setting it should nevertheless be possible to apply such optimisations. For this to be possible, we have to ensure that any outstanding dynamic checks are properly preserved.

The solution we develop throughout the remainder of this paper tries to tackle all these challenges. Discussions of individual design decisions try to relate their impact on the challenges identified here.

3 SaC$_\lambda$

SaC$_\lambda$ is a functional language inspired by SaC, comprising only the bare essentials of SaC that are needed for a functional array language; its syntax closely resembles that of SaC. However, we have modified it to a λ-calculus style, in order to ease comprehension by a functional-programming audience.

$$
\begin{array}{lll}
\mathit{Program} & \Rightarrow & \left\lceil \mathit{FunId}\ = \lambda\, \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*}\ .\ \mathit{Expr}\ ;\ \right\rfloor^{*} \\
 & & \quad \textbf{main}\ = \mathit{Expr}\ ; \\
 & & \\
\mathit{Expr} & \Rightarrow & [\ \left\lceil \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*} \right\rceil\ ] \\
 & & \mid\ \mathit{FunId}\ (\ \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*}\ ) \\
 & & \mid\ \mathit{Prf}\ (\ \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*}\ ) \\
 & & \mid\ \textbf{if}\ \mathit{Id}\ \textbf{then}\ \mathit{Expr}\ \textbf{else}\ \mathit{Expr} \\
 & & \mid\ \textbf{let}\ \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*} = \mathit{Expr}\ \textbf{in}\ \mathit{Expr} \\
 & & \mid\ \mathit{Const} \\
 & & \mid\ \mathit{Id} \\
 & & \\
\mathit{Prf} & \Rightarrow & \textbf{shape}\ \mid\ \textbf{dim}\ \mid\ \textbf{sel}\ \mid\ \textbf{modarray} \\
 & & \mid\ \textbf{add\_SxS}\ \mid\ \textbf{add\_SxA}\ \mid\ \textbf{add\_AxS}\ \mid\ \textbf{add\_AxA} \\
 & & \mid\ \textbf{eq\_SxS}\ \mid\ \textbf{eq\_SxA}\ \mid\ \textbf{eq\_AxS}\ \mid\ \textbf{eq\_AxA} \\
 & & \mid\ \ldots
\end{array}
$$

Fig. 1. The syntax of SaC$_\lambda$

Note that the version of SaC$_\lambda$ used in this paper differs from versions presented in earlier papers: We focus on built-in primitive functions rather than higher-level constructs like the WITH-loop [3]. Figure 1 shows the syntax of SaC$_\lambda$. A program consists of a set of mutually recursive function definitions and a designated main expression. Essentially, expressions are either constants, variables or function applications. Since SaC, at present, neither supports higher-order functions nor nameless functions, all abstractions (function definitions) are explicitly user-defined. Function applications are written in C-style, *i.e.*, with parentheses around arguments, rather than around entire applications of functions. To simplify the formal presentation in later sections of this paper, we restrict the arguments to be identifiers rather than arbitrary expressions. However, a transformation of unrestricted programs into this restricted form is straight-forward.

SaC$_\lambda$ provides a few built-in array operations, referred to as primitive functions (Prf). Among these are `shape` and `dim` for computing an array's shape and dimensionality (rank), respectively. A selection operation, `sel`, is also provided; it takes two arguments: an index vector, specifying the element to be selected,

and an array from which to select. As its dual, SaC$_\lambda$ provides a `modarray` operation which computes a new array from an existing one by altering a single element only; it takes three arguments: a template array, the index position at which the result array is supposed to be different from the template array and the value to which the referenced element of the array is to be set. These basic array operations are complemented by element-wise extensions of arithmetic and relational operations, such as *addition* (`add`) and *equality* (`eq`), respectively, with similar semantics to those of APL and J. We differentiate between two different kinds of arguments to these binary operations: Array arguments, denoted by the letter `A`, and scalar arguments, represented by the letter `S`. This leads to a total of four versions of each binary operation, one for each combination of argument classes. To differentiate between these, we use the suffices `SxS`, `SxA`, `AxS` and `AxA`.

The versions defined on arguments of the same kind, *i.e.*, `SxS` and `AxA`, compute by applying the operation element wise to each pair of corresponding elements of the two arguments. Binary operations with non-matching argument classes, *i.e.*, `SxA` and `AxS`, compute by applying the operation to each element of the array argument and the single scalar argument.

We can formalize the semantics of SaC$_\lambda$ by a standard big-step operational semantics for λ-calculus-based applicative languages as defined in several textbooks, *e.g.*, [14]. As a unified representation for n-dimensional arrays, we use pairs of vectors $<[\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m]>$ where the vector $[\ s_1, \ldots, s_n]$ denotes the shape of the array, *i.e.*, its extent with respect to the n individual axes, and the vector $[\ d_1, \ldots, d_m]$ contains all elements of the array in a row-major linearized form.

The first two evaluation rules of Figure 2 show how scalars as well as vectors are transformed into the internal representation. The rule Vect requires that all elements have the same shape to ensure shape consistency in the overall result.

The semantics of `let` expressions is formalized by the third rule. We use the standard substitution function $e[v/\alpha]$ which substitutes all free occurrences of variable α within the expression e by an expression v.

The next two rules describe the semantics of function definition and application. To allow for recursive function definitions, we use an explicit fix-point operator `fix` in conjunction with the substitution function described above. For each function definition, rule LetRec substitutes all applied occurences within the remainder of the program by an application of `fix` to the function name and definition. The corresponding definition of function application is given by rule Ap. It differs from the standard rule for applicative languages only by the additional substitution of recursive function applications within the function body by an explicit `fix` operator. We use $e[v_i/\alpha_i]_{i=1}^{n}$ to denote the sequence of substitutions $e[v_1/\alpha_1] \cdots [v_n/\alpha_n]$.

Rule Main gives the semantics of the `main` expression of a program. A formal definition of conditionals in SaC$_\lambda$is given by the rules IfTrue and IfFalse.

The next four rules formalize the semantics of the main primitive operations on arrays: `dim`, `shape`, `sel` and `modarray`. There are two aspects of the Sel rule to be observed: first, we require the selection index to be of the same length as

the shape of the array to be selected from. This ensures scalar values as results. Second, the selection index must be within the bounds of the array argument, *i.e.*, each element i_j of the index vector needs to be non-negative and less than the corresponding element s_j of the shape vector of the array argument. Finally, the selection requires a transformation of the index vector into a scalar offset l into the linearized form of the array. The sum of products used here reflects the row-major linearization we have chosen.

The rule MODARRAY imposes the same restrictions on the index vector and the array argument of the `modarray` operation. Additionally, we require that the third argument to `modarray` is a scalar value. This is to ensure that the

$$\textsc{Const} : \frac{}{n \rightarrow < [\,], [n] >}$$

$$\textsc{Vect} : \frac{\forall i \in \{1, \ldots, n\} : e_i \rightarrow < [\ s_1, \ldots, s_m], [\ d_1^i, \ldots, d_p^i] >}{[\ e_1, \ldots, e_n] \rightarrow < [\ \mathrm{n},\ s_1, \ldots, s_m], [\ d_1^1, \ldots, d_p^1, \ldots, d_1^n, \ldots, d_p^n] >}$$

$$\textsc{Let} : \frac{\begin{array}{c} e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] > \\ e_b[< [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >/\alpha] \rightarrow < [\ s'_1, \ldots, s'_k], [\ d'_1, \ldots, d'_l] > \end{array}}{\texttt{let}\ \alpha\ \texttt{=}\ e\ \texttt{in}\ e_b \rightarrow < [\ s'_1, \ldots, s'_k], [\ d'_1, \ldots, d'_l] >}$$

$$\textsc{LetRec} : \frac{p[\texttt{fix}\ f\ \lambda\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{.}e/f] \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\begin{array}{l}\texttt{f = } \lambda\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{.}e \\ p\end{array} \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}$$

$$\textsc{Ap} : \frac{\begin{array}{c} \forall i \in \{1, \ldots, n\} : e_i \rightarrow < [\ s_1^i, \ldots, s_{n_i}^i], [\ d_1^i, \ldots, d_{m_i}^i] > \\ e[< [\ s_1^i, \ldots, s_{n_i}^i], [\ d_1^i, \ldots, d_{m_i}^i] >/\alpha_i]_{i=1}^n[\texttt{fix}\ f\ \lambda\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{.}e/f] \\ \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] > \end{array}}{\begin{array}{c}\texttt{fix}\ f\ \lambda\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{.}e\texttt{(}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} \\ \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >\end{array}}$$

$$\textsc{Main} : \frac{e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{main = } e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}$$

$$\textsc{IfTrue} : \frac{e_p \rightarrow <\ [\,],\ [\ \texttt{true}] >\ \ e_t \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{if}\ e_p\ \texttt{then}\ e_t\ \texttt{else}\ e_e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}$$

$$\textsc{IfFalse} : \frac{e_p \rightarrow <\ [\,],\ [\ \texttt{false}] >\ \ e_e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{if}\ e_p\ \texttt{then}\ e_t\ \texttt{else}\ e_e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}$$

$$\textsc{Dim} : \frac{e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{dim(}\ e\texttt{)} \rightarrow < [\,], [n] >}$$

$$\textsc{Shape} : \frac{e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{shape(}\ e\texttt{)} \rightarrow < [\ n], [\ s_1, \ldots, s_n] >}$$

Fig. 2. An operational semantics for $\textsc{SaC}_\lambda$

$$\textsc{Sel} : \frac{\begin{array}{c} iv \rightarrow < [\,n], [\,i_1, \ldots, i_n] > \\ e \rightarrow < [\,s_1, \ldots, s_n], [\,d_1, \ldots, d_m] > \end{array}}{\texttt{sel(}\ iv\texttt{,}\ e\texttt{)} \rightarrow < [\,], [\,d_l] >}$$
$$\text{where } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) + 1$$
$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k$$

$$\textsc{ModArray} : \frac{\begin{array}{c} iv \rightarrow < [\,n], [\,i_1, \ldots, i_n] > \\ e_d \rightarrow < [\,s_1, \ldots, s_n], [\,d_1, \ldots, d_m] > \\ e_v \rightarrow < [\,], v > \end{array}}{\texttt{modarray(}\ iv\texttt{,}\ e_d\texttt{,}\ e_v\texttt{)} \rightarrow < [\,s_1, \ldots, s_n], [\,d'_1, \ldots, d'_m] >}$$
$$\text{where } d'_l = \begin{cases} v & \text{if } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) + 1, \\ d_l & \text{otherwise.} \end{cases}$$
$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k$$

$$\textsc{Add\_SxS} : \frac{\begin{array}{c} e_1 \rightarrow < [\,], d_1 > \\ e_2 \rightarrow < [\,], d_2 > \end{array}}{\texttt{add\_SxS(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\,], [\,d_1 + d_2] >}$$

$$\textsc{Add\_AxS} : \frac{\begin{array}{c} e_1 \rightarrow < [\,s_1, \ldots, s_n], [\,d^1_1, \ldots, d^1_m] > \\ e_2 \rightarrow < [\,], d > \end{array}}{\texttt{add\_AxS(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\,s_1, \ldots, s_n], [\,d^1_1 + d, \ldots, d^1_m + d] >}$$

$$\textsc{Add\_AxA} : \frac{\begin{array}{c} e_1 \rightarrow < [\,s_1, \ldots, s_n], [\,d^1_1, \ldots, d^1_m] > \\ e_2 \rightarrow < [\,s_1, \ldots, s_n], [\,d^2_1, \ldots, d^2_m] > \end{array}}{\texttt{add\_AxA(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\,s_1, \ldots, s_n], [\,d^1_1 + d^2_1, \ldots, d^1_m + d^2_m] >}$$

Fig. 2. (*continued*)

`modarray` operation results in a homogeneous array, *i.e.*, that the replaced value and replacing value are of the same shape.

Element-wise extensions of standard operations, such as the arithmetic and relational operations, are demonstrated by the example of the rules for addition (`add_SxS`, `add_AxS` and `add_AxV`). We have left out the rule for the `SxA` variant, as it is symmetrical to the `AxS` variant.

Whereas `add_SxS` and `add_AxS` can be applied to any pair of scalar values or an array of arbitrary shape as first argument and any scalar value as second argument, respectively, we require the arguments of `add_AxA` to be of the same shape.

4 Representing Constraints as Contracts

In this Section, we will discuss different approaches to representing constraints as explicit contracts in SAC$_\lambda$. As a first step, we have to identify the implicit constraints of the primitive functions built into SAC$_\lambda$. To begin with, consider the following application of the binary primitive function `add_AxS`:

```
...let R = add_AxS( A, v)
in...
```

where A and v are defined in the surrounding context. From rule ADD_AxS in Figure 2, we can deduce that the second argument needs to evaluate to a scalar value. Thus, the above application of add_AxS has the following constraint:

1. v is required to evaluate to a scalar value

This constraint is an example for a constraint on the dimensionality of an array, *i.e.*, static knowledge of the dimensionality of the second argument to add_AxS suffices to evaluate this constraint statically.

For an application of add_AxA like

```
...let R = add_AxA( A, B)
in...
```

where A and B are given by the surrounding context, we get a different class of constraints. As rule ADD_AxA in Figure 2 shows, the following constraint needs to hold in order for the application to be evaluated:

2. A and B evaluate to values of the same shape

In contrast to constraint 1 above, constraint 2 requires static shape knowledge of both arguments, more precisely, static knowledge of shape equalities.

Similarly, constraints for add_SxS, add_SxA and the remaining binary operations can be derived. Finally, we need to derive constraints for sel and modarray operations. As an example, consider the following applications of sel and modarray:

```
...let B = modarray( A, iv, v)
in let w = sel( B, iv)
in...
```

where A, iv and v are defined in the surrounding context.

As for the previous examples, by looking at the semantic rules defined in Figure 2, we can deduce the following implicit constraints for the application of modarray:

3. the length of iv needs to match the dimensionality of A
4. iv is required to be non-negative
5. each element of iv needs to be smaller than the corresponding value of the shape vector of A
6. v should be a scalar value

For the application of sel we get:

7. the length of iv is required to match the dimensionality of B
8. iv needs to be non-negative
9. each element of iv is required to be smaller than the corresponding value of the shape vector of B

Constraints 3 and 7 are constraints on the shape of arguments of a primitive function, similar to constraint 2 shown in the previous example. However, constraints 4 and 8, and 5 and 9 are constraints that depend on the value of an argument, *i.e.*, these can only be statically decided if the value of the corresponding arguments are known at compile time.

Having identified the constraints for the built-in functions of SaC$_\lambda$, as a next step we need to encode these as SaC$_\lambda$ expressions. In the following, we will explore the design space and discuss three different means to express contracts in SaC$_\lambda$.

Reusing Existing Primitive Functions. A straightforward approach would be to directly encode the constraints using existing SaC$_\lambda$ built-in functions. For example, constraint 1 can be encoded by the following expression:

```
eq_SxS( dim( v), 0)
```

However, although a direct encoding complies with our goal to only require minimal implementation work, it has its drawbacks. Firstly, using existing built-in functions requires potentially multiple nested expressions. As an example, consider an implementation of constraint 2:

```
all( eq_AxA( shape( A), shape(B)))
```

where `all` is the element-wise logical *and* operation on arrays. Here, expressing one constraint as an explicit contract requires four primitive operations. If performed for each primitive function in a program, this leads to a major code explosion.

Secondly, using existing primitive functions may lead to a non-terminating code transformation. In the example above, `eq_AxA` requires its two arguments to be of the same shape. Thus, inserting a contract to ensure that two expressions evaluate to arrays of the same shape yields the same constraint again.

Finally, as discussed in Section 2, it is essential to be able to give suitable feedback about which constraints remain to be checked at run-time to the programmer. However, by reusing existing primitive functions to express contracts, the latter become indistinguishable from user written code.

Tailor-Made Functions. To circumvent code explosion and to make contracts easily distinguishable from user-written expressions, we chose to express the constraints of each primitive function via dedicated built-in functions. These functions are tailor-made to express contracts, so they can be designed in such a way that they do not have any constraints apart from those they assert. This resolves the potential termination problem of a corresponding code transformation.

As an example, we could define a new primitive function `ensure_add_AxA` which ensures that all constraints for an application of `add_AxA` hold. The contract for constraint 2 can then be encoded as:

```
ensure_add_AxA( A, B)
```

where `A` and `B` are the arguments to the corresponding application of `add_AxA`. However, using a single function to encode a set of constraints might hinder partial evaluation. As an example, consider using a single primitive function, *e.g.*,

`ensure_modarray`, for applications of the `modarray` operation. Here, different constraints require different levels of static knowledge. For example, constraint 3 requires only static knowledge of the dimensionality of one argument, whereas constraint 6 requires static knowledge of the shape and even value of one argument. Thus, although in principal some constraints could be statically decided, using this coarse grained approach, a partially static decision cannot be expressed in the code. The partial evaluator can only either evaluate all constraints statically, or leave all checks for evaluation at run-time.

Fine-Grained Tailor-Made Functions. To combine the strengths of both approaches presented so far, without adopting their weaknesses, we propose a third approach. To limit code explosion and ease the extraction of suitable feedback, we use dedicated primitive functions to express contracts. As we noted, this ensures the termination of a corresponding code transformation. In contrast to the second approach, we define one primitive function for each constraint instead of defining one function per set of constraints. This allows us to statically evaluate parts of the constraints of a primitive function.

To put the third approach into action, we need to add the following five additional primitive functions. For constraints 1 and 3, we add:

`is_scalar`, which evaluates to `true` if its argument is a scalar value and to `false` otherwise.

The shape-dependent constraint 2 can be expressed using:

`same_shape`, which evaluates to `true` if its two arguments have the same shape and to `false` otherwise.

To express constraints 3 and 7, 4 and 8, and 5 and 9, respectively, we add the following primitive functions:

`shape_matches_dim`, which evaluates to `true` if the length of its first argument, *i.e.*, the shape at position 0, matches the dimensionality of the second, otherwise it evaluates to `false`.

`non_neg_val` which evaluates to `true` if all elements of its first argument are non-negative, otherwise to `false`.

`val_lt_shape`, which evaluates to `true` if each element of the first argument is smaller than the corresponding element of the shape of the second argument, otherwise it evaluates to `false`.

Using the above functions, we can now insert explicit contracts for implicit constraints of primitive functions into the code.

5 Inserting Contracts for Primitive Functions

So far, we have discussed different means to express the contracts resulting from constraints of primitive functions in SaC$_\lambda$. However, to make use of these contracts, we furthermore need to insert them into the code. A viable solution with respect to the challenges laid out in Section 2 thereby needs to meet the following criteria:

1. The contracts need to safeguard the corresponding primitive functions such that the primitive function is only evaluated if the contracts hold. Otherwise, the program should terminate with an error.
2. Contracts should be accessible to the existing partial evaluator and optimisations. In particular, knowledge gained by evaluating contracts should be propagated as far as possible.
3. Optimisations should profit from knowledge gained by contracts, *i.e.*, optimisations should not need to be aware of the constraints of primitive functions.

In the following, we will present three different approaches to insert contracts into the code and discuss their suitability with respect to the above criteria.

Contracts by Conditionals. As a first approach, we consider wrapping applications of primitive functions into conditionals. For example, the following SaC$_\lambda$ expression:

```
...let R = add_AxS( A, v)
in...
```

where `A` and `v` are defined in the surrounding context, can be transformed into:

```
...let R = if ( is_scalar( v)) then add_AxS( A, v)
                                else ⊥
in...
```

We use the symbol $\bot$, denoted *bottom*, to represent an explicit program termination. In the above example, the application of `add_AxS` is only evaluated if the application of `is_scalar` to `v` evaluates to `true`, *i.e.*, if `v` evaluates to a scalar value. Otherwise, the program terminates. Thus, using conditionals clearly fulfils the first criterion.

However, with respect to the second criterion, the above solution is not optimal. The result of evaluating the predicate of the conditional, `is_scalar(v)`, is only available within the scope of the conditional, *i.e.*, its `then` and `else` branch. Optimisations on, or partial evaluation of, expressions containing `v` within the body of the surrounding `let` expression cannot exploit this additional knowledge. Although this situation could be mitigated by wrapping the entire `let` expression instead of the application of `add_AxS` inside the conditional, such a transformation is not straightforward.

Weaving Contracts into the Dataflow. Another way to introduce contracts into the code is to weave them into the dataflow. That is, instead of using the tailor-made contract functions as predicates, redefine those functions so that, if the constraint holds, they return the argument for which they assert the constraint; otherwise, they terminate the evaluation. Thus, if all constraints hold, the program evaluates as expected. If one of the constraints is violated, the evaluation terminates with an error.

As an example, reconsider the application of `add_AxS` given above. Using the dataflow approach, the code can be extended by contracts as follows:

```
...let v' = is_scalar( v)
in let R  = add_AxS( A, v')
in...
```

In the above example, the application of `is_scalar` guards the consecutive application of `add_AxS`. Therefore, like the previous approach, weaving constraints into the dataflow fulfils the first criterion. Moreover, other than the first approach, it fulfils the second criterion, as well. As the result of evaluating `v` and asserting the constraint is bound to a new identifier `v'`, we now have an explicit handle to the additional knowledge gained by evaluating the contract. To make this knowledge available within the body of the surrounding `let` expression, it suffices to substitute all occurrences of `v` in the body by `v'`, a well understood and simple transformation.

Finally, we have to assess criterion three. Consider the following excerpt from a SaC$_\lambda$ expression:

```
...let B = modarray( A, iv, v)
in let w = sel( iv, B)
in...
```

where `A`, `iv` and `v` are defined in the surrounding context. In the above code, we first compute a new array `B` by replacing the value at position `iv` with `v`. In the consecutive application of `sel`, we then select this value again and bind the result to the identifier `w`. Under the assumption that the applications of `modarray` and `sel` are safe, we know statically that `w` equals `v` and therefore can simplify the above code to the following expression:

```
...let B = modarray( A, iv, v)
...let w = v
in...
```

If `B` is not referenced in the body of the surrounding `let` expression, we can furthermore remove the application of `modarray`, as its result is not needed anymore.

The above example might look artificial but in the setting of SaC, *i.e.*, a language with a high level of abstraction and the presence of sophisticated optimisations, expressions like the one presented here are surprisingly common.

For the above transformation to be semantic preserving, we have to ensure that the constraints of the applications of `modarray` and `sel` hold. For example, if `iv` is an invalid index into array `A`, the non-optimised version will fail, whereas the fully optimised version would succeed, thereby computing the wrong result.

As demanded by the third criterion, inserting explicit contracts should allow us to blindly apply this optimisation, without checking any constraints. Consider the following transformation:

```
...let v1  = is_scalar( v)
in let iv1 = non_neg_val( iv)
in let iv2 = shape_matches_dim( iv1, A)
in let iv3 = val_lt_shape( iv2, A)
in let B   = modarray( A, iv3, v1)
in let iv4 = non_neg_val( iv3)
in let iv5 = shape_matches_dim( iv4, B)
in let iv6 = val_lt_shape( iv5, B)
in let w   = sel( iv6, B)
in...
```

In this setting, our primitive optimisation cannot be applied, as the `sel` operation uses an index vector different from the one used in the `modarray` operation. However, many of the above contracts can be statically evaluated. Firstly, we statically know that `iv1` is non-negative. Therefore, `iv2` and `iv3` are non-negative, as well. Thus, we can deduce that the application of `non_neg_val` to `iv3` is the identity function. Secondly, we know that the shape of `B` is equal to the shape of `A`, as `B` is computed from `A` using a shape-preserving `modarray` operation. Therefore, the second application of `shape_matches_dim` and `val_lt_shape`, respectively, return the identity of their first argument. By combining this static knowledge, we can deduce that `iv6` equals `iv3` and simplify the above code as follows:

```
...let v1  = is_scalar( v)
in let iv1 = non_neg_val( iv)
in let iv2 = shape_matches_dim( iv1, A)
in let iv3 = val_lt_shape( iv2, A)
in let B   = modarray( A, iv3, v1)
in let w   = sel( iv3, B)
in ...
```

Now, our simple optimisation can be applied again. On first glance this is safe, as the explicit contracts guard the consecutive applications of `modarray` and `sel`. However, by replacing the application of `sel` by `v1`, we might remove the last reference to `B`, which will, should `iv3` not be used anywhere else, turn the `modarray` operation and the corresponding contracts into dead code. With these contracts being eliminated the optimised program will produce the wrong result if the definition of `B` becomes dead code.

As the above example shows, just weaving the contracts into the dataflow does not suffice to meet the third criterion.

Using Explicit Evidence. To allow for a naive application of optimisations like the one shown above without sacrificing semantic soundness, we have to ensure that inserted contracts cannot be removed as a result of an optimisation, as long as the result of a corresponding application of a primitive function contributes to the overall result. For the above example, this means that we have to ensure that the contracts for the application `sel` stay intact. More precisely, we have to ensure that the contracts of `sel` are not removed, even if no further use of `B` exists.

To achieve this, we propose the use of *explicit evidence* that a contract is fulfilled. We then explicitly check this evidence before using the result of an application of a primitive function. Thereby, the contracts will remain intact, even if the computation as such has been removed, as long as the computation's result is used.

We implement this by extending the primitive functions used for expressing contracts with a further Boolean return value. If a contract holds, the primitive function additionally returns `true`. Otherwise the evaluation terminates. To tie the evidence to the result of an application of a primitive function, we introduce a further primitive function:

`after_guard`, which takes two or more arguments, is the identity in its first argument, if all consecutive arguments evaluate to `true`; otherwise, it terminates evaluation.

As an example of its use, consider the extended version of the above example:

```
...let v1,  e1 = is_scalar( v)
in let iv1, e2 = non_neg_val( iv)
in let iv2, e3 = shape_matches_dim( iv1, A)
in let iv3, e4 = val_lt_shape( iv2, A)
in let B       = modarray( A, iv3, v1)
in let B1      = after_guard( B, e1, e2, e3, e4)
in let iv4, e5 = non_neg_val( iv3)
in let iv5, e6 = shape_matches_dim( iv4, A)
in let iv6, e7 = val_lt_shape( iv5, A)
in let w       = sel( iv6, B1)
in let w1      = after_guard( w, e5, e6, e7)
in...
```

Here, the result of the application of `modarray` is tied to the corresponding contracts using the evidence returned by the contracts and an application of `after_guard`. Similarly, the result of the application of `sel` is guarded. Applying the same reasoning as in the previous approach, we can reduce the number of contracts as follows:

```
...let v1,  e1 = is_scalar( v)
in let iv1, e2 = non_neg_val( iv)
in let iv2, e3 = shape_matches_dim( iv1, A)
in let iv3, e4 = val_lt_shape( iv2, A)
in let B       = modarray( A, iv3, v1)
in let B1      = after_guard( B, e1, e2, e3, e4)
in let w       = sel( iv3, B1)
in let w1      = w
in...
```

Note that the second application of `after_guard` has been replaced by its first argument, as we statically know that the corresponding evidence evaluates to `true`.

In the above setting, our simple optimisation cannot be applied as long as the evidence of the application of `modarray` cannot be statically evaluated to `true`. However, if the remaining `after_guard` vanishes, thereby enabling our optimisation, we can be sure that the optimisation can safely be applied as all contracts have been statically evaluated. Thus, this extended dataflow representation fulfils all three critera.

6 A Formal Definition

In the following, we describe the approach developed in the previous sections more formally. We first give the semantics of the added primitive functions. As a second step, we formalise the transformation scheme that inserts contracts

$$\text{SAME\_SHAPE} \;:\; \frac{\begin{array}{r} e_1 \rightarrow < [\,s_1, \ldots, s_i], [\,d_1^1, \ldots, d_k^1] > \\ e_2 \rightarrow < [\,s_1, \ldots, s_i], [\,d_1^2, \ldots, d_l^2] > \end{array}}{\begin{array}{rl} & < [\,s_1, \ldots, s_i], [\,d_1^1, \ldots, d_k^1] >, \\ \texttt{same\_shape(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow & < [\,s_1, \ldots, s_i], [\,d_1^2, \ldots, d_l^2] >, \\ & < [], \text{true} > \end{array}}$$

$$\text{AFTER\_GUARD} : \frac{e \rightarrow v \quad \forall i \in \{1, \ldots, n\} : e_i \rightarrow < [], \text{true} >}{\texttt{after\_guard(}\ e\texttt{,}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} \rightarrow v}$$

Fig. 3. Semantic rules for the additional built-in functions `same_shape` and `after_guard`

into the code. Using these definitions, we finally sketch out a proof that the transformation is semantic-preserving.

Due to space limitations, we concentrate in our presentation on the function `add_AxA` and the corresponding contracts. However, an extension to all primitive functions is straightforward.

Figure 3 shows the semantic rules for the primitive functions `same_shape` and `after_guard`. As described informally in Section 5, `same_shape` is the identity on its first two arguments and returns `true` as its third result only if the shapes of its arguments match. Otherwise, the evaluation gets stuck and ultimately fails. Similarly, `after_guard` is the identity on its first argument only if all other arguments evaluate to `true`.

To formally describe the insertion of contracts discusses in Section 5, we use the code transformation scheme C sketched out in Figure 4. Basically, it replaces all occurrences of `add_AxA` with the corresponding guarded expression. Note that Id'_A, Id'_B, Id_E, Id'_R denote fresh variables that have no free occurrences within the body expression e.

In order to propagate knowledge gained from evaluating contracts, we furthermore substitute the arguments of `same_shape` by its results within the body expression. This substitution is performed using the environment E. Whenever we need to substitute an identifier, we add a pair (Id, Id') to the environment, where Id is the identifier to be substituted and Id' denotes the substitute. Rule Id performs this substitution. The *lookup* function is defined in the usual way:

$$\text{(ADD\_AXA)}\ \ \text{C}\left[\!\!\left[\begin{array}{l}\texttt{let}\ Id_R\ \texttt{=}\ \texttt{add\_AxA(}\ Id_A\texttt{,}\ Id_B\texttt{)} \\ \texttt{in}\ e\end{array}\right.\!\!\!\left.,\ E\right]\!\!\right]$$

$$\rightsquigarrow \begin{array}{l}\texttt{let}\ Id'_A\texttt{,}\ Id'_B\texttt{,}\ Id_E\ \texttt{=}\ \texttt{same\_shape(}\ \text{C}[\![Id_A,\ E]\!]\texttt{,}\ \text{C}[\![Id_B,\ E]\!]) \\ \texttt{in let}\ Id'_R\ \texttt{=}\ \texttt{add\_AxA(}\ Id'_A\texttt{,}\ Id'_B\texttt{)} \\ \texttt{in let}\ Id_R\ \texttt{=}\ \texttt{after\_guard(}\ Id'_R\texttt{,}\ Id_E\texttt{)} \\ \texttt{in}\ \text{C}[\![e,\ E +\!+ < (Id_A, Id'_A), (Id_B, Id'_B) >]\!]\end{array}$$

$$\text{(ID)}\ \ \text{C}[\![Id,\ E]\!] \rightsquigarrow \textit{lookup}(\ \text{Id, E})$$

Fig. 4. Transformation scheme for inserting explicit contracts for applications of the primitive function `add_AxA`

lookup(Id, E) returns the most recent substitute for Id in E, if one exists. Otherwise, it returns Id.

Using these definitions, we can now sketch out a proof that the code transformation C is semantic-preserving.

Theorem 1. C *is sound with respect to the semantics of* SaC$_\lambda$

Proof. From the semantics definition in Figures 2 and 3 we can see that it suffices to show that

$$\frac{e_1 \rightarrow < s,\ d_1 >\ \ e_2 \rightarrow < s,\ d_2 >}{\begin{array}{l}\texttt{let a, b, e = same\_shape(}\ e_1\texttt{,}\ e_2\texttt{)} \\ \texttt{in let r = add\_AxA(a, b)} \qquad \rightarrow < s,\ d_1 + d_2 > \\ \texttt{in after\_guard(r, e)}\end{array}}$$

For the application of `after_guard` *we know that*

$$\frac{< s,\ d_1 + d_2 > \rightarrow < s,\ d_1 + d_2 >\ < \texttt{[]}, \texttt{true} > \rightarrow < \texttt{[]}, \texttt{true} >}{\texttt{after\_guard(}\ < s,\ d_1 + d_2 >, < \texttt{[]}, \texttt{true} >\texttt{)} \rightarrow < s,\ d_1 + d_2 >}$$

Similarly, for `add_AxA` *we can deduce*

$$\frac{< s,\ d_1 > \rightarrow < s,\ d_1 >\ < s,\ d_2 > \rightarrow < s,\ d_2 >}{\texttt{add\_AxA(}\ < s,\ d_1 >, < s,\ d_2 >\texttt{)} \rightarrow < s,\ d_1 + d_2 >}$$

Finally, we can deduce from rule SAME_SHAPE *that:*

$$\frac{e_1 \rightarrow < s,\ d_1 >\ \ e_2 \rightarrow < s,\ d_2 >}{\texttt{same\_shape(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < s,\ d_1 >, < s,\ d_2 >, < \texttt{[]}, \texttt{true} >}$$

By applying the standard semantics of `let`*, we yield the required deduction. q.e.d.*

7 Constraint Resolution by Partial Evaluation

We have implemented the transformation sketched out in Figure 4 as part of our research compiler `sac2c`. First evaluations have shown that the presented representation integrates well with our existing optimisations. In essence, only few extensions to some of our standard optimisations, *e.g.*, Constant Folding, were required in order to be able to statically resolve a large proportion of the contracts. Most of the other optimisations integrated in our compiler, such as Common Subexpression Elimination and Dead Code Removal (for an overview see [3]), contribute to the constraint resolution without any modification.

The key drivers behind these optimisations appear to be our existing shape and dimensionality inference mechanisms. To gather static shape knowledge, we use the shape inference that is part of the SaC type-system [3]. In short, the SaC type-system statically infers array shapes where possible but resorts to subtyping-based type-weakening where a fully static approach would be undecidable. We enrich this information with shape and dimensionality equalities inferred by further symbolic analyses [15,16].

Table 1. Quantitative results of inserting explicit contracts into the Livermore FORTRAN kernels

		loop1	loop2	loop3	loop4	loop5	loop6	loop7	loop9	loop10	loop11	loop12	loop13	loop14	loop16
no-opt	all	985	274	125	655	224	217	890	310	377	184	299	472	885	695
	val.-dep.	596	174	84	412	144	140	546	196	236	120	188	296	536	430
opt	all	125	103	95	123	98	95	200	194	202	89	101	136	166	146
	val.-dep.	87	72	67	86	69	67	137	133	138	63	71	94	114	101
resolved in %	all	87.3	62.4	24.0	81.2	56.3	56.2	77.5	37.4	46.4	51.6	66.2	71.2	81.2	79.0
	val.-dep.	85.4	58.6	20.2	79.1	52.1	52.1	75.0	32.1	41.5	47.5	62.2	68.2	78.7	76.5

Due to the tight integration of these techniques, it is difficult to attribute the effects to individual optimisations. Even a quantification of the overall effect is intricate for our current prototype, as the constraint insertion is not implemented naïvely but utilises the inferred type information already.

Keeping these limitations in mind, we have performed a quantitative analysis of the number of inserted and resolved explicit contracts using the *Livermore Loops* [17] to gain preliminary insights into the effectiveness of our approach.

The Livermore Loops are a collection of FORTRAN kernels from real-world numerical applications which have been used in a performance comparison between SISAL and FORTRAN [18]. For our experiments, we have used the SAC implementation that is available as part of the compiler distribution. For all measurements, we have used revision 15670 of the developer version of `sac2c`. To measure the number of inserted contracts, we have compiled the different kernels using the compiler options `-noOPT -doDCR -doINL -doDFR -maxspec 0 -check c`. These disable all but the bare essential optimisations, *i.e.*, DEAD CODE REMOVAL, FUNCTION INLINING and DEAD FUNCTION REMOVAL. Furthermore, we have disabled function specialisation to minimize the static shape knowledge available to the type system. The last option enables the insertion of constraints as explicit contracts as described in this paper. To measure the number of primitive functions used for explicit contracts in the intermediate code after optimisations, we have used the built-in optimisation statistics of `sac2c`. The results are given in Table 1, aggregated over all primitive functions (first row) and only those that depend on values (second row).

In a second run, we have compiled the same programs using the compiler options `-maxlur 3 -check c`. The first option limits the LOOP UNROLLING optimisation built into `sac2c` to ensure that the loops contained in the source code are not eliminated. The number of primitive functions remaining in the code after all optimisations is shown in the third and fourth row of Table 1.

As an indicator for the effectiveness of our approach, we have computed the difference between the first and second run in percent (cf. row five and six of Table 1). As can be seen, we were able to resolve an average of 62.7% of the inserted contracts statically. Taking only the value-dependent contracts into account, we are still able to resolve an average of 59.2% statically.

Currently, our symbolic optimisations are focused on shape and dimensionality information. We therefore would have expected the difference between the overall resolution ratio and that for value-dependent contracts to be more pronounced. However, as our non-naïve insertion process eliminates many shape- and dimensionality-dependent contracts before inserting them, the results are biased. An in-depth quantitative analysis of our approach remains future work.

8 Related Work

Our work is similar to dependent type systems like *indexed types* [5] or the approaches used in ZPL [7] and CHAPEL [8], in that we try to prove shape and value dependent constraints statically. However, instead of using a sophisticated type system and imposing restrictions on the use of indices, we utilise explicit constraints and partial evaluation. Furthermore, our approach allows dynamic checks to remain in the generated code when static analysis does not permit all constraints to be satisfied statically.

Hybrid type checking [19] also facilitates static constraint satisfaction, while supporting dynamic checks when those cannot be satisfied. In contrast to the work presented here, the author proposes to drive type inference as far as possible, and only introduce dynamic checks when it gets stuck. Our approach starts out with blindly inserted dynamic checks and then tries to evaluate these statically. This allows us to use existing partial evaluation techniques instead of enriching our type inference system.

A similar approach has been proposed by Xu for the lazy functional language HASKELL [20]. ESC/HASKELL uses symbolic evaluation combined with counter-example guided unrolling to statically prove user-defined pre- and post-conditions. In contrast to the approach presented here, ESC/HASKELL mainly focusses on debugging whereas we additionally aim at enhancing program runtimes and simplifying the implementation of optimisations.

The idea of explicit evidence as used by our dataflow representation is used in [21] as well. However, the authors deal with a low-level byte-code that has already been verified and enriched with dynamic checks by a compiler. In their paper, they focus on retaining these checks across program optimisations, whereas we furthermore exploit contracts for static guarantees and try to minimise the number of runtime checks.

9 Conclusions

This paper demonstrates how compiler-inserted contracts, in conjunction with partial evaluation and other optimisation techniques, can be used to obtain static conformity guarantees similar to those that can be expressed by dependent types or variants thereof.

The effectiveness of our approach arises from insertion of carefully designed evidence-gaining predicates into the data flow and from use of evidence guards on the function results. Due to the explicit encoding as part of the program

code, this evidence is accessible to the existing partial evaluator and further optimisations. In particular, we can use the existing optimisations to remove many redundant conformity checks and are able to substantially simplify the implementation of several symbolic optimisations within the compiler itself.

As the experience from our prototypical implementation shows, the proposed approach can be implemented with minimal effort. The presented transformation to insert contracts is straight-forward and contract resolution comes nearby for free. Only minor extensions to the Constant Folding implementation are required. Apart from minimising the implementation effort, reusing existing optimisations comes with a further benefit: Future enhancements to existing optimisations as well as the addition of further optimisations automatically benefit contract checking.

In this paper, we focus our presentation on checking domain constraints for built-in functions. However, we believe the approach is far more versatile. The concept of explicit evidence-carrying variables equally well applies to contracts for user-defined functions. This brings the properties of our system close to that of more strongly typed systems based on various forms of dependent types: for many programs, we can give static soundness guarantees with respect to certain domain requirements. In those cases where we cannot give these guarantees, we can clearly identify the program parts where unresolved constraints remain. Then, it is up to the user to decide whether further program optimisation should be applied or the dynamic contract checks should remain.

It remains as future research to investigate whether such a general purpose optimisation mechanism is capable of resolving more complex constraints in an effective way. In particular, it would be interesting to compare its effectiveness with that obtained by dedicated resolution systems such as Epigram [22].

Acknowledgements

We would like to thank the attendees and anonymous referees of IFL 2007 for their valuable feedback.

References

1. International Standards Organization: Programming Language APL, Extended. ISO N93.03, ISO (1993)
2. Iverson, K.E.: J Introduction and Dictionary. In: J release. 3rd edn. (1996)
3. Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13(6), 1005–1059 (2003)
4. Bernecky, R.: Reducing computational complexity with array predicates. In: APL 1998: Proceedings of the APL98 conference on Array processing language, pp. 39–43. ACM Press, New York (1998)
5. Zenger, C.: Indexed types. Theoretical Computer Science 187(1-2), 147–165 (1997)
6. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: POPL 1999, pp. 214–227. ACM Press, New York (1999)

7. Chamberlain, B., Choi, S.E., Lewis, E., Lin, C., Snyder, L., Weathersby, W.: ZPL: A Machine Independent Programming Language for Parallel Computers. IEEE Transactions on Software Engineering 26(3), 197–211 (2000)
8. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. International Journal of High Performance Computing Applications, 291–312 (2007)
9. Meyer, B.: Eiffel: The Language. Prentice Hall, Englewood Cliffs (1990)
10. Meyer, B.: Applying design by contract. Computer 25(10), 40–51 (1992)
11. Kramer, R.: iContract - The Java(tm) design by contract(tm) tool. In: TOOLS 1998, p. 295. IEEE Computer Society, Los Alamitos (1998)
12. Karaorman, M., Holzle, U., Bruno, J.: jContractor: A reflective java library to support design by contract. In: Proceedings Reflection 1999, The Second International Conference on Meta-Level Architectures and Reflection, Santa Barbara, CA, USA, pp. 19–21. University of California at Santa Barbara (1999)
13. Ploesch, R.: Design by contract for python. In: APSEC 1997: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference, Washington, DC, USA, p. 213. IEEE Computer Society, Los Alamitos (1997)
14. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
15. Trojahner, K., Grelck, C., Scholz, S.B.: On Optimising Shape-Generic Array Programs using Symbolic Structural Information. In: Horváth, Z., Zsók, V. (eds.) IFL 2006. LNCS, vol. 4449, pp. 1–18. Springer, Heidelberg (2007)
16. Bernecky, R.: Shape cliques. ACM SIGAPL Quote Quad 35(3), 7–17 (2007)
17. McMahon, F.H.: The livermore fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA (1986)
18. Cann, D., Feo, J.: SISAL versus FORTRAN: a comparison using the livermore loops. In: Supercomputing 1990: Proceedings of the 1990 ACM/IEEE conference on Supercomputing, Washington, DC, USA, pp. 626–636. IEEE Computer Society, Los Alamitos (1990)
19. Flanagan, C.: Hybrid type checking. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 245–256. ACM, New York (2006)
20. Xu, D.N.: Extended static checking for haskell. In: Haskell 2006: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, pp. 48–59. ACM, New York (2006)
21. Menon, V.S., Glew, N., Murphy, B.R., McCreight, A., Shpeisman, T., Adl-Tabatabai, A.R., Petersen, L.: A verifiable ssa program representation for aggressive compiler optimization. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Charleston, South Carolina, USA, pp. 397–408. ACM, New York (2006)
22. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. 14(1), 69–111 (2004)

Author Index