Lecture Notes in Artificial Intelligence 5437

Edited by R. Goebel, J. Siekmann, and W. Wahlster

Subseries of Lecture Notes in Computer Science

Dietmar Seipel Michael Hanus
Armin Wolf (Eds.)

Applications of Declarative Programming and Knowledge Management

17th International Conference, INAP 2007
and 21st Workshop on Logic Programming, WLP 2007
Würzburg, Germany, October 4-6, 2007
Revised Selected Papers

Springer

Volume Editors

Dietmar Seipel
Universität Würzburg
Institut für Informatik
Am Hubland, 97074 Würzburg, Germany
E-mail: seipel@informatik.uni-wuerzburg.de

Michael Hanus
Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Christian-Albrechts-Platz 4, 24118 Kiel, Germany
E-mail: mh@informatik.uni-kiel.de

Armin Wolf
Fraunhofer FIRST
Kekuléstr. 7, 12489 Berlin, Germany
E-mail: Armin.Wolf@first.fraunhofer.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): I.2.4, I.2, D.1.6

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN	0302-9743
ISBN-10	3-642-00674-4 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-00674-6 Springer Berlin Heidelberg New York

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12624225 06/3180 5 4 3 2 1 0

Preface

This volume contains a selection of papers presented at the 17th International Conference on Applications of Declarative Programming and Knowledge Management INAP 2007 and the 21st Workshop on Logic Programming WLP 2007, which were held jointly in Würzburg, Germany, during October 4–6, 2007.

Declarative programming is an advanced paradigm for the modeling and solving of complex problems. This specification method has become more and more attractive over the last years, e.g., in the domains of databases, for the processing of natural language, for the modeling and processing of combinatorial problems, and for establishing knowledge–based systems for the Web.

The INAP conferences provide a forum for intensive discussions of applications of important technologies around logic programming, constraint problem solving, and closely related advanced software. They comprehensively cover the impact of programmable logic solvers in the Internet society, its underlying technologies, and leading-edge applications in industry, commerce, government, and societal services.

The Workshops on Logic Programming are the annual meeting of the Society for Logic Programming (GLP e.V.). They bring together researchers interested in logic programming, constraint programming, and related areas like databases and artificial intelligence. Previous workshops have been held in Germany, Austria and Switzerland.

The topics of the selected papers of this year's joint conference concentrated on three currently important fields: constraint programming and constraint solving, databases and data mining, and declarative programming with logic languages.

During the last couple of years a lot of research has been conducted on the usage of declarative programming for *databases and data mining*. Reasoning about knowledge wrapped in rules, databases, or the Web allows one to explore interesting hidden knowledge. Declarative techniques for the transformation, deduction, induction, visualization, or querying of knowledge, or data mining techniques for exploring knowledge have the advantage of high transparency and better maintainability compared to procedural approaches.

The problem when using knowledge to find solutions for large industrial tasks is that these problems have an exponential complexity, which normally prohibits the fast generation of exact solutions. One method which has made substantial progress over the last years is the *constraint programming* paradigm. The declarative nature of this paradigm offers significant advantages for software engineering both in the implementation and in the maintenance phase. Different interestings aspects are under discussion: how can this paradigm be improved or combined with known, classical methods; how can practical problems be modeled as constraint problems; and what are the experiences of applications in really large industrial planning and simulation tasks?

Another area of active research is the *extension* of the *logic programming paradigm* and its integration with other programming concepts. The successful extension of logic programming with constraints has been already mentioned. Other extensions intend to increase the expressivity of logic languages by including new logical constructs, such as contextual operators or temporal annotations. The integration of logic programming with other programming paradigms has been mainly investigated for the case of functional programming. This combination is beneficial from a software engineering point of view: well–known functional programming techniques to improve the structure and quality of the developed software, e.g., types, modules, higher–order operators, or lazy evaluation, can be also used for logic programming in an integrated language.

The two conferences INAP 2007 and WLP 2007 were jointly organized at the University of Würzburg, Germany, by the following institutions: the University of Würzburg, the Society for Logic Programming (GLP e.V.), and the Fraunhofer Institute for Computer Architecture and Software Technology (FhG First). We would like to thank all authors who submitted papers and all conference participants for the fruitful discussions. We are grateful to the members of the Program Committee and the external referees for their timely expertise in carefully reviewing the papers, and we would like to express our thanks to the Institute for Bioinformatics of the University of Würzburg for hosting the conference.

September 2008

Dietmar Seipel
Michael Hanus
Armin Wolf

Organization

Program Chair

Dietmar Seipel	University of Würzburg, Germany

Program Committee of INAP

Sergio A. Alvarez	Boston College, USA
Oskar Bartenstein	IF Computer Japan, Japan
Joachim Baumeister	University of Würzburg, Germany
Henning Christiansen	Roskilde University, Denmark
Ulrich Geske	University of Potsdam, Germany
Parke Godfrey	York University, Canada
Petra Hofstedt	Technical University of Berlin, Germany
Thomas Kleemann	University of Würzburg, Germany
Ilkka Niemelä	Helsinki University of Technology, Finland
David Pearce	Universidad Rey Juan Carlos, Madrid, Spain
Carolina Ruiz	Worcester Polytechnic Institute, USA
Dietmar Seipel	University of Würzburg, Germany (Chair)
Osamu Takata	Kyushu Institute of Technology, Japan
Hans Tompits	Technical University of Vienna, Austria
Masanobu Umeda	Kyushu Institute of Technology, Japan
Armin Wolf	Fraunhofer FIRST, Germany
Osamu Yoshie	Waseda University, Japan

Program Committee of WLP

Slim Abdennadher	German University Cairo, Egypt
Christoph Beierle	Fern University Hagen, Germany
Jürgen Dix	Technical University of Clausthal, Germany
Thomas Eiter	Technical University of Vienna, Austria
Tim Furche	University of München, Germany
Ulrich Geske	University of Potsdam, Germany
Michael Hanus	Christian Albrechts University Kiel, Germany (Chair)
Petra Hofstedt	Technical University of Berlin, Germany
Sebastian Schaffert	Salzburg Research, Austria
Torsten Schaub	University of Potsdam, Germany
Sibylle Schwarz	University of Halle, Germany
Dietmar Seipel	University of Würzburg, Germany

Michael Thielscher	Technical University of Dresden, Germany
Hans Tompits	Technical University of Vienna, Austria
Armin Wolf	Fraunhofer FIRST, Berlin, Germany

Local Organization

Dietmar Seipel	University of Würzburg, Germany
Joachim Baumeister	University of Würzburg, Germany

External Referees for INAP and WLP

Stephan Frank

Martin Grabmüller

Table of Contents

Invited Talk

Constraints

Databases and Data Mining

Extensions of Logic Programming

System Demonstrations

A Guide for Manual Construction of Difference-List Procedures

Ulrich Geske[1] and Hans-Joachim Goltz[2]

[1] University of Potsdam
ugeske@uni-potsdam.de
[2] Fraunhofer FIRST, Berlin
goltz@first.fraunhofer.de

Abstract. The difference-list technique is an effective method for extending lists to the right without using the append/3 procedure. There exist some proposals for automatic transformation of list programs into difference-list programs. However, we are interested in a construction of difference-list programs by the programmer, avoiding the need of a transformation. In [9] it was demonstrated, how left-recursive procedures with a dangling call of append/3 can be transformed into right-recursion using the unfolding technique. For some types of right-recursive procedures, the equivalence of the accumulator technique and difference-list technique was shown and rules for writing corresponding difference-list programs were given. In the present paper, improved and simplified rules are derived which substitute the formerly given ones. We can show that these rule allow us to write difference-list programs which supply result-lists that are either constructed in top-down -manner (elements in append order) or in bottom-up manner (elements in inverse order) in a simple schematic way.

1 What Are Difference-Lists?

On the one hand, lists are a useful modelling construct in logic programming because of their non-predefined length; on the other hand, the concatenation of lists by `append(L1,L2,L3)` is rather inefficient because it copies the list `L1`. To avoid the invocation of the `append/3` procedure, an alternative method is to use incomplete lists of the form `[el`$_1$`,...el`$_k$`|Var]`, in which the variable `Var` describes the tail of the list that is not yet specified completely. If there is an assignment of a concrete list for the variable `Var` in the program, it will result in an efficient (physical) concatenation of the first list elements `el`$_1$`,...el`$_k$ and `Var` without copying these elements. This physical concatenation does not consist in an extra-logical replacement of a pointer (a memory address), but is a purely logical operation since the reference to the list `L` and its tail `Var` was already created by their specifications in the program.

From a mathematical point of view, the difference of the two lists `[el`$_1$`,...,el`$_k$`|Var]` and `Var` denotes the initial piece `[el`$_1$`,...,el`$_k$`]` of the complete list. For example, the difference `[1,2,3]` arises from the lists `[1,2,3|X]`, and `X` or arises from `[1,2,3,4,5]` and `[4,5]` or may arise `from [1,2,3,a]` and `[a]`. The first-mentioned possibility is the most general representation of the list difference

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 1–20, 2009.

[1, 2, 3]. Every listmay be presented as a difference-list. The empty list can be expressed as a difference of the two lists L and L, the list List is the difference of the list List and the empty list ([]).

The combination of the two list components $[el_1,...,el_k|Var]$ and Var in a structure with the semantics of the list difference is called a "difference-list". Since such a combination is based on the possibility of specifying incomplete lists, which is always possible, the Prolog standard does not provide any special notation for this combination. A specification of a difference-list from the two lists L and R may be given by a list notation [L,R] or by the use of an operator, e.g. L-R or L\R (the symbol used must be defined as operator in the concrete Prolog system) or as two separate arguments separated by a comma.

In practical programming, the concatenation of lists is very frequently expressed using an invocation of the append/3 procedure. The reason for this may be the inadequate explanations for the use of incomplete lists and of the difference-list technique that uses such incomplete lists. Very different explanations for the use of difference-lists and very different attitudes to them can be found in well-known manuals and textbooks on Prolog.

2 Difference-Lists in the Literature

2.1 Definition of Difference-Lists

The earliest extended description of difference-lists was given by Clark and Tärnlund in [7]. They illustrated the difference-list or d-list notation by the graphical visualization of lists as used in the programming language LISP. One of their examples is shown in Figure 1.

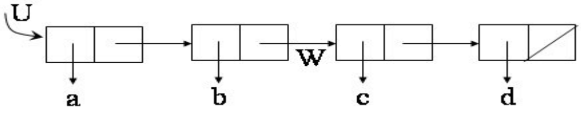


Fig. 1. Visualization of the difference-list <U, W> = [a, b | W] - W = [a, b]

In this example, the list [a,b,c,d] can be considered as the value of the pointer U. A sublist of U, the list [c,d], could be represented as the value of the pointer W. The list [a,b] between both pointers can be considered as the difference of both lists. In [7], this difference is denoted by the term <U,W>, which is formed from both pointers. Other authors denote it by a term U\W [14]. We prefer to use the representation U-W in programs, as is done in [13]. A difference-list could be defined as in [7]:

Definition (difference-list, in short: d-list or dl):

$$\text{d-list}(\langle U, W\rangle) \leftrightarrow U = W \vee \exists X \exists V \{U = [X \mid V] \ \& \ \text{element}(X) \ \& \ \text{d-list}(V, W)\} \quad (1)$$

The pair `<U,W>` represents the difference of the lists `U` and `W` and is therefore called a difference-list (d-list or dl for short). The difference includes all the elements X of the list, which begins at the pointer `U` and ends at pointer `W` to a list or to `nil`.

In a LISP-like representation, the value of the difference of list `U` and `W` can be constructed by replacing the pointer to `W` in `U` by the empty list. Let us introduce the notation `Val(<U,W>)` for the value of the term that represents a difference-list `<U, W>`. Of course, the unification procedure in Prolog does not perform the computation of the value, just as it does not perform arithmetic computations in general.

Clark and Tärnlund have given and proved the definition of construction and concatenation of difference-lists [7]:

Definition (concatenation of difference-lists):

$$\text{d-list}(\langle U, W\rangle) \leftrightarrow \exists V\{\text{d-list}(\langle U, V\rangle) \;\&\; \text{d-list}(V, W)\} \qquad (2)$$

If <[b, c, d | Z], Z> is a difference-list with V=[b, c, d | Z] and W=Z, then the difference-list <U, W>= <[a,b,c,d | Z], Z> can be constructed, where X=a and U=[a,b,c,d | Z]. A concatenation of the difference-lists `<U,V>=<[a,b|X],X>` and `<V,W>=<X,nil>` results in the difference-list `<U,W>=<[a,b,c,d],nil>` as soon as `X` is computed as `[c, d]`. The concept of concatenation of difference-lists will feature prominently in our rules for programming d-list programs.

2.2 Automatic Transformation of List Programs

Several authors have investigated automatic transformations of list programs into difference-list programs [1, 10, 11, 15]. Mariott and Søndergaard [11] have analyzed the problem intensively and proposed a stepwise algorithm for such a transformation. The problem is to ensure that such a transformation is safe. Table 1shows that the semantics of programs may differ when changing programs from list notation to difference-list notation. Essentially, the absence of an occur-check in Prolog systems is due to the different behaviour of programs in list and d-list notation. But Table 1 also shows that if the occur-check were available, different semantics of list procedures and the corresponding d-list procedures might occur.

Table 1. Semantics of a program in list and difference-list representation

Notation	Intended meaning	Prolog without occur-check	Prolog with occur-check

list	`qs([],[]).`
	`?-qs([],[a]).`
	`> no`
	`?-qs([],[a|Z]).`
	`> no`
		`qs([],[]).`
		`?-qs([],[a]).`
		`> no`
		`?-qs([],[a|Z]).`
		`> no`
			`qs([],[]).`
			`?-qs([],[a]).`
			`> no`
			`?-qs([],[a|Z]).`
			`> no`

dl	`qs([],L-L).`
	`?-qs([],[a|Z]-Z).`
	`> no`
	
	`?-qs([],[a|Z]-Y).`
	`> no`
		`qs([],L-L).`
		`?-qs([],[a|Z]-Z).`
		`> Z=[a,a,a…]` (*yes*)
		`?-qs([],[a|Z]-Y).`
		`> Y=[a|Z]`
		`yes`
			`qs([],L-L).`
			`?-qs([],[a|Z]-Z).`
			`> no`
			
			`?-qs([],[a|Z]-Y).`
			`> Y=[a|Z]` (*yes*)

Looking for conditions for a safe automatic generation of difference-lists, Marriot and Søndergaard [11] begin by considering of difference-terms:

Definition (difference-term):
A difference-term is a term of the form <T,T′>, where T and T′ are terms. (3)
The term T′ is the delimiter of the difference-term. The denotation of <T,T′> is the term that results when every occurrence of T′ in T is replaced by nil.

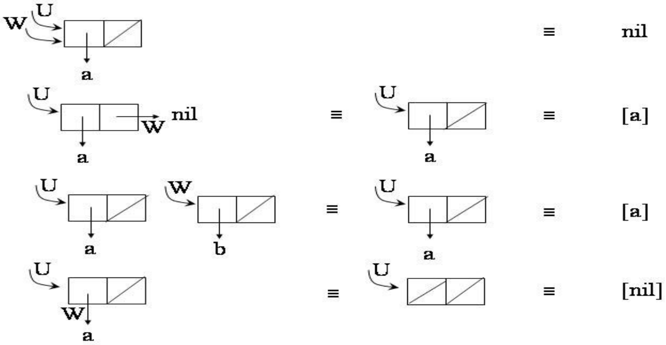


Fig. 2. Difference-trees (visualization of examples of [14]) (nil, /, and [ ] are synonyms for the empty list)

Figure 2 shows some examples of the difference-terms given in [11]. If both U and W point to the list [a], the difference-term <U,W> represents the empty list, i.e. Val(<[a],[a]>)=[]. If U points to a list T and W points to nil, the tail of T, then <U,W> represents T (the pointer to W in T is replaced by nil), i.e. Val(<T,nil>)=T. The difference-term <T,X> is a generalization of the difference-term <[a],[b]> if X is not part of (does not occur in) T. In cases where the second part of the difference-term does not occur in the first part, there is nothing to change in the first part of the difference-term (no pointer is to be replaced by nil) and the first part of the difference-term is the value, i.e. Val(<[a],[b]>)=[a] and Val(<T,X>)=T. In cases where W does not point to a list but to an element of a list, a similar replacement action can be performed to get the value of the difference-term. In <U,W>=<[a],a>, W points to the list element a. If the pointer to a is replaced by nil, Val(<[a],a>)=[nil] is obtained. A similar but more complex example is given by the difference-term <g(X,X),X>. All pointers to X have to be replaced by nil, which results overall in g(nil,nil).

In the context of the transformation of list procedures into corresponding d-list procedures, two difference-terms should unify if the corresponding terms unify. This property is valid for simple difference-terms [11]:

Definition (simple difference-term):
A difference-term is simple, iff it is of the form <T,nil> or of the form <T,X>, where X represents a variable. (4)

Definition (simple difference-list; closed difference-list):
A difference-list is a difference-term that represents a list. It is closed, iff the list that it represents is closed (i.e., the difference-term is of the form <[T1, …, Tn], Tn>). The difference-list is simple, if Tn is nil or a variable. (5)

The definition of difference-lists allows that Tn could be a pointer not only to a list but to any term. Thus, each closed list can be represented as a difference-list, but not every difference-list is the representation of a closed list. The automatic transformation of lists into difference-lists is safe if the generated difference-list is simple and T, Tn in <T, Tn> have non-interfering variables.

Table 2 should remind us that the problem of different semantics of programs is not a special problem for difference-lists. Also, in pure list processing, the programmer must be aware that bindings of variables to themselves may occur during unification and may give rise to different behaviour depending on the programming system.

In [11] the authors investigate conditions for safe transformation of list programs into d-list programs and present an algorithm for this transformation.

2.3 Difference-Lists in Prolog Textbooks

Clocksin has described the difference-list technique in a tutorial [4] based on his book "Clause and Effect" [3] as "*probably one of the most ingenious programming techniques ever invented yet neglected by mainstream computer science*". By contrast, Dodd´s opinion [8] concerning a pair of variables *(All, Temp)* for the result *All* of a list operation and the used accumulator *Acc* is less emphatic: "*Some people find it helpful to think of* All *and* Temp *as a special data structure called difference list. We prefer to rest our account of the matter on the logical reading of the variable as referring to any possible tail, rather than to speak of 'lists with variable tails'*".

Table 2. Different semantics of programs

Intended meaning	Prolog without occur-check	Prolog with occur-check

`append([],L,L).`
`?-append([],[a|Z],Z)`
`> no`
	`append([],L,L).`
	`?-append([],[a|Z],Z)`
	`> Z=[a,a,a,…]` (*yes*)
		`append([],L,L).`
		`?-append([],[a|Z],Z)`
		`> no`

In the classical Prolog textbook of **Clocksin and Mellish** [5], a certain relationship between accumulator technique and difference-list technique is defined: "With difference list we also use two arguments (comment: as for the use of accumulators), but with a different interpretation. The first is the *final result, and the second argument is for a hole in the final result where further informatio*n can be put".

O'Keefe [1] uses the same analogy: **"**A common thing to do in Prolog is to carry around a partial data structure and some of the holes in it. To extend the data structure, you fill in a hole with a new term which has some holes of its own. Difference lists are a special case of this technique where the two arguments we are passing around are the position in a list**".** The represented program code for a subset relation corresponds in its structure to the bottom-up construction of lists. This structure is identified as "difference-list" notation.

Clark&McCabe [6] denote a difference-list as a difference pair. It is used as a substitute for an append/3 call. The effects are described by means of the procedure for the list reversal. If the list to be reversed is described by [X|Y], the following logical reading exists for the recursive clause: “[X|Y] has a reverse represented by a difference pair [Z, Z1] if Y has a reverse represented by a difference pair [Z, [X|Z1]] ". ...suppose that [Z, [X|Z1]] is [[3,2,1,0], [1,0]] representing the list [3,2], ...then [Z, Z1] of is [[3,2,1,0], [0]] representing the list [3,2,1]". ... you will always get back a most general difference pair representation of the reverse of the list”. Besides this example, no further guidelines for programming with difference-lists are given.

<table>
<tr><th>Flattening of lists using difference-lists</th><th>Flattening of lists using accumulators</th></tr>
<tr><td>flatten_dl([X|Xs], R-Acc):-</td><td>flatten([X|Xs], Acc, R) :-</td></tr>
<tr><td>   flatten_dl(X,R-R1),</td><td>   flatten(Xs, Acc, R1),</td></tr>
<tr><td>   flatten_dl(Xs,R1-Acc).</td><td>   flatten(X, R1, R).</td></tr>
<tr><td>flatten_dl(X,[X|Xs]-Xs) :-</td><td>flatten(X, Xs, [X|Xs]) :-</td></tr>
<tr><td>   atom(X),not X=[].</td><td>   atom(X),not X=[].</td></tr>
<tr><td>flatten_dl([],Xs-Xs).</td><td>flatten([],Xs, Xs).</td></tr>
</table>

Fig. 3. Comparison of list flattening in the difference-list and accumulator technique [14]

A detailed presentation of the difference-list technique is given by **Sterling and Shapiro** [14]. They stress the tight relationship between accumulator technique and difference-list technique: “...Difference-lists generalize the concept of accumulators" and “... Programs using difference-lists are sometimes structurally similar to programs written using accumulators". A comparison of a flatten_dl/2 procedure, that uses difference-lists and a flatten/3 procedure, that uses the accumulator technique is given (the order of the arguments X,Y in a difference-list X-Y is normally, without semantic consequences, exchanged in the accumulator format representation Y,X).

Unfortunately, the conclusion drawn from this comparison is not a construction rule for the use of difference-lists, but the (partly wrong) remark that the difference between both methods “…is the goal order in the recursive clause of flatten". In fact, the two recursive flatten/3 calls in the flatten/3 procedure can be exchanged without changing the result. Instead of trying to formalize the technique, it is “mystified": “Declaratively the (comment: flatten_dl/2)... is straightforward. ... The operational behaviour... is harder to understand. The flattened list seems to be built by magic. ... The discrepancy between clear declarative understanding and difficult procedural understanding stems from the power of the logical variable. We can specify logical relationships implicitly, and leave their enforcement to Prolog. Here the concatenation of the difference-lists has been expressed implicitly, and is mysterious when it happens in the program." ([14], S. 242). On the other hand, the authors mention that by using a difference-list form of the call to append/3 and an unfolding operation in respect of the definition of append/3, an append/3-free (difference-list) program can be derived.

Colho&Cotta [2] compare two versions for sorting a list using the quicksort algorithm (Figure 4). They attempt, as in [14], to explain the difference between the two programs with the help of the difference-list interpretation of append/3: the complete difference S-X results from the concatenation of the differences S-Y and Y-X.

Maier&Warren [12] mention the effect of avoiding append/3 calls using difference-lists. They point out that the property of difference-lists is to decompose lists in one step into a head and a tail, each of arbitrary length. This property is useful in natural language processing, where phrases consist of one or more words. In order to make this processing append/3-free, word information, e.g. the information that “small" is an adjective, is represented in the difference-list format: `adjective([[small|Rest]-Rest])` instead of using the list format `adjective([small])`.

Quicksort	
Definition with append/3	Definition with difference-lists
`q_sort1([X|Xs], S) :-`	`q_sort2([X|Xs], S-S1) :-`
`   split(X,Xs,A,B),`	`   split(X,Xs,A,B),`
`   q_sort1(A,S1),`	`   q_sort2(A,S-[X|S2]),`
`   q_sort1(B,S2),`	`   q_sort2(B,S2-S1).`
`   append(S1, [X|S2], S).`	
`q_sort1([],[]).`	`q_sort2([],X-X).`
`?-  q_sort1([1,2,3],X).`	`?- q_sort2([1,2,3],X-[]).`
`X = [1,2,3]`	`X = [1,2,3]`

Fig. 4. Comparison of algorithms for quicksort [2]

Dodd [8] explains how left-recursive programs, in particular those that use `append/3` as last call, are transformed into more efficient programs using the accumulator technique. For this purpose, he develops a formalism based on functional compositions that generates a right-recursive procedure from a left-recursive procedure in six transformation steps. The effects achieved by the accumulator technique are shown by a great number of problems, and the relationship to the difference-list technique is mentioned, but there is no systematic approach for using accumulators or difference-lists.

The different techniques available for generating lists can be reduced to two methods which are essential for list processing: top-down and bottom-up generation of structures. In [9], we have shown that the difference-list notation could be considered syntactic sugar since it can be derived from procedures programmed using the accumulator technique by a simple syntactic transformation. However, the difference-list notation is not only syntactic sugar, since a simple modelling technique using difference-lists instead of additional arguments or accumulators can offer advantages constructing result lists with certain order of elements, efficiently. Knowledge of the (simple) programming rules for difference-list procedures can make it easier to use difference-lists and promotes their use in programming.

3 Top-Down and Bottom-Up Construction of Lists

The notions of top-down and bottom-up procedures for traversing structures like trees are well established. We will use the notions top-down construction of lists and bottom-up construction of lists in this paper to describe the result of the process of building lists with a certain order of elements in relation to the order in which the elements are taken from the corresponding input list.

An input list may be, e.g., [2,4,3,1]. A top-down construction of the result list [2,4,3,1] is given if the elements are taken from left to right from the input list and put into the constructed list in a left to right manner. If the elements are taken from the input list by their value and put into the result list from left to right, the list [1,2,3,4] will be (top-down) constructed.

Definition (top-down (TD) construction of lists):
The order el$_1$’ – el$_2$’ of two arbitrary elements el$_1$’, el$_2$’ in the constructed list corresponds to the order el$_1$ – el$_2$ in which the two elements el$_1$, el$_2$ are taken from the input term (perhaps a list or a tree structure). (6)

Definition (bottom-up (BU) construction of lists):
The order el$_2$’ – el$_1$’ of two arbitrary elements el$_1$’, el$_2$’ in the constructed list corresponds to the order el$_1$ – el$_2$ in which the two elements el$_1$, el$_2$ are taken from the input term (perhaps a list or a tree structure). (7)

A bottom-up construction of the result list [1,3,4,2] is given if the elements are taken from left to right from the input list and put into the constructed list in a right-to-left manner. If the elements are taken from the input list by their magnitude and put into the result list from right to left, the list [4,3,2,1] will be (bottom-up) constructed.

<table>
<tr><th></th><th>Top-Down construction of lists</th><th>Bottom-Up construction of lists</th></tr>
<tr><td>Use of calls of append/3</td><td>copy_td_ap([],L,L).
copy_td_ap([X|Xs],A,R):-
   copy_td_ap(Xs,A,RR),
   append([X],RR,R).
?- copy_td_ap([2,1],[],X).
X = [2, 1]</td><td>copy_bu_ap([],L,L).
copy_bu_ap([X|Xs],A,R):-
   append([X],A,RR),
   copy_bu_ap(Xs,RR,R).
?- copy_bu_ap([2,1],[],X).
X = [1, 2]</td></tr>
<tr><td>Calls of append/3 unfolded</td><td>copy_td([],L,L).
copy_td([X|Xs],A,[X|R]):-
   copy_td(Xs,A,R).
?- copy_td([2,1],[],X).
X = [2, 1]</td><td>copy_bu([],L,L).
copy_bu([X|Xs],A,R):-
   copy_bu(Xs,[X|A],R).
?- copy_bu([2,1],[],X).
X = [1, 2]</td></tr>
<tr><td>Use of differ-ence-lists</td><td>dl_copy_td([],L-L).
dl_copy_td([X|Xs],[X|R]-A):-
   dl_copy_td(Xs,R-A).
?- dl_copy_td([2,1],X-[]).
X = [2, 1]</td><td>dl_copy_bu([],L-L).
dl_copy_bu([X|Xs],R-A):-
   dl_copy_bu(Xs,R-[X|A]).
?- dl_copy_bu([2,1],X-[]).
X = [1, 2]</td></tr>
</table>

Fig. 5. Some examples of top-down and bottom-up construction of lists

Figure 5 shows some definitions for the tasks of copying lists. The procedures copy_td_ap/3 and copy_td/3 are procedures that construct the result list in a top-down manner. The list elements of the result list are supplied in the same order as elements of the input list. The procedures copy_bu_ap/3 and copy_bu/3 are procedures that construct the result list in a bottom-up manner. The elements of the result list occur in an inverse order in relation to the elements of the input list. The procedures copy_td/3 and copy_bu/3 result by unfolding the append/3-calls in copy_td_ap/3 and copy_bu_ap/3. We use symbol X for a single list element. All other symbols denote lists: A accumulator list, L arbitrary lists, R, S result list, RR, T, Z temporary result lists, Xs remaining list.Construction in a top-down manner means that the first element chosen from the input list will be the first element of the result list, which is recursively extended to the right. Construction in a bottom-up manner means that the first element chosen from the input list will be the last element in the result list, which is recursively extended to the left. In each case, the extension of the result parameter is performed by a cons operation. This is the (only) basic operation, which is used implicitly by append/3, for extending a list `List` by a single element `X` to `[X|List]`. The single element X extends the list List always to the left by this operation. The question is where to insert this cons operation? To answer this question, different complex problems are investigated in the next sections.

4 Analysis of Procedure Pattern

Comparison of the append-free procedures for copying lists in Figure 5 allows us to assume that there may exist the following rules for top-down and bottom-up definitions: *A top-down construction of lists would occur if the next element X from the input list were part of the result list* `[X|RestResult]` *or* `[X|Rest]-A` *in the clause head, and a bottom-up construction of lists would occur if the next element X from the input list were inserted in the result list* `[X|Accu]` *or* `R-[X|Accu]` *in the clause body.* The definition of q_sort2/2 in Figure4 is proof of the contrary. Moreover, a more detailed investigation is needed because the comparison of the procedures presented in Figure 5 may be misleading. The difference-list procedures in this figure are the result of a simple syntactic reformulation of procedures that use unfolding of append/3-calls occuring in the original procedures. The question arises and will be answered in the following if this kind of unfolding preserves the semantics of the original procedure.

<table>
<tr><th colspan="3">Inverse of a list</th></tr>
<tr><td>Procedure definition using append/3</td><td>Procedure definition using accumulator technique</td><td>Procedure definition using difference-lists</td></tr>
<tr><td><pre>rv([],[]).
rv([X|Xs],R) :-
  append(RR,[X],R),
  rv(Xs,RR).
?-rv([a,b],[a|X]).
Stack overflow</pre></td><td><pre>a_rv([],L,L).
a_rv([X|Xs],A,R) :-
  a_rv(Xs,[X|A],R).

?-a_rv([a,b],[],[a|X]).
no</pre></td><td><pre>d_rv([],L-L).
d_rv([X|Xs],R-A):-
  d_rv(Xs,R-[X|A]).

?-d_rv([a,b],[a|X]-[]).
no</pre></td></tr>
</table>

Fig. 6. Semantics of different definitions for the inverse of a list

Figure 6 shows definitions and invocations of different versions for computing the inverse of a list. These procedures correspond to constructing the result list in a bottom-up manner. The elements of the result list have an inverse order compared with the order of the elements of the input list. The difference-list version is a syntactic

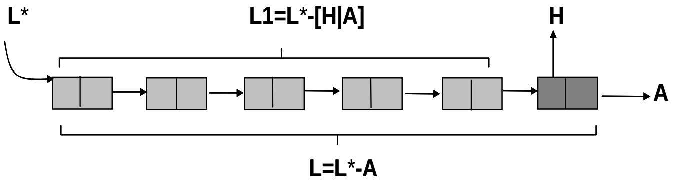


Fig. 7. Visual representation of appending list L1 and list [H|A]

variant of the accumulator version of the procedure. The relationship between these two procedures and the procedure rv/2 which uses a call to append/3 is given by unfolding the append/3 call. While the first argument for the append/3 call in Figure 6 is not a list with one element (as in copy-td_ap/3 – see also Figure 5) but a list with an indefinite number of elements, this situation is considered more detailed in Figure 7.

Sublists in Figure 7 are denoted in their difference-list notation. The result of the procedure call append (L1,[H],L) may be described as L1=L*-[H|A] or as L=L*-A, where A may be an empty or nonempty list. If L and L1 in rv/2 are replaced by these relationships and if the append/3 call is deleted, then the difference-list definition d_rv/2 results. This means that the difference-list definition and the accumulator-technique definition for the inverse of a list may be derived from the original definition (using append/3) by unfolding the append/3 call. However the semantics of the two derived procedures is different from the semantics of the original procedure definition. The result of the analysis so far is that unfolding of append/3-calls generally fails to supply a semantically equivalent difference-list notation. We therefore try to avoid append/3 calls in the original procedure definition. An alternative might be to append lists in the original procedure by corresponding use of an append procedure in difference-list notation.

Figure 8 shows a difference-list definition dl_a/2 for appending lists derived from the original append/3 definition by simply exchanging the order of the second and third argument and collecting both arguments into one term, a difference-list.

Definition for append/3	Difference-list definition for appending lists
append([],L,L). append([X|Xs],A,[X|RR]):- append(Xs,A,RR). ?- append([a,b,c],[], L). L = [a,b,c]	dl_a([],L-L). dl_a([X|Xs],[X|RR]-A):- dl_a(Xs,RR-A). ?- dl_a([a,b,c],L-[]). L = [a,b,c]

Fig. 8. Comparison of the definitions append/3 and dl_a/2 for appending lists

Inside the definition of dl_a/2 there occurs a dl_a/2-call, i.e. an append-like call that performs an undesired copying of list elements. In order to design the difference-list definition for appending lists free of calls to dl_a/2, the difference-list notation will be extended to all arguments of dl_a/2. Fig. 9 represents the concatenation of lists A and T to a composed list, e.g. Z.

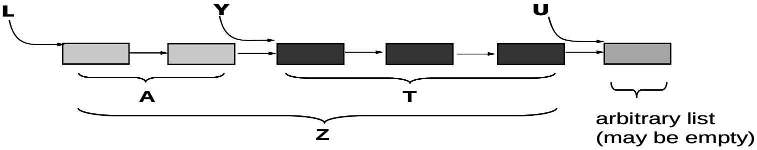


Fig. 9. Difference-list interpretation of the concatenation of two lists

List A can be represented as a difference-list in the form L-Y. The list T has the difference-list format Y-[] or more generally: Y-U, where U is assumed to be an arbitrary further list (perhaps the empty list []). Correspondingly, the composed list Z may be represented as the difference-list L-U or, in case U=[ ], as L-[]. In this way, the procedure for the list concatenation may be represented by:

```
  dl_append(L-Y, Y-U, L-U).
```

Fig. 10. Difference-list notation for the concatenation of two lists

```
  ?- dl_append(L-L,L-U,L-U).
   yes
```

Fig. 11. Treatment of the end condition of list concatenation

From the existence of the difference-lists A−B and B−C,
the existence of the difference-list A−C can be deduced (and vice versa).

Rule 1: Composition of difference-lists

```
   cons(Element,[Element|Temp]-Temp).
```

Fig. 12. Definition of the cons operation

The end condition append([],Y,Y) of the append/3 procedure is contained in dl_append/3 since it may be Y=L, i.e. L-L = [ ] (see Figure 11). From the definition of dl_append/3 the programming rule for difference-lists results (Rule 1). It is important that this rule has a logical reading. The rule does not contain regulations for the order of the proofs (i.e. the order of procedure calls). This rule could be considered the key for the desired construction rule for difference-lists. We have to include in our considerations the cons operation [X|L], which is needed to transfer an element from the input list to the result list. But [X|L] is in a format that does not suit the above rule. For example, in the bottom-up procedure dl_copy_bu/2 (see also Figure 5) for copying list elements, the difference-list R-A for the result is composed partly from the difference-list R-[X|A]. Let us introduce a temporary variable Z and substitute Z=[X|A], as shown in Figure 13. If we reformulate this call of the unification as a call cons(X,Z-A) of a special cons/2 procedure (Figure 12) we are able to apply the

<table>
<tr><th colspan="3">Copying of lists</th></tr>
<tr><td>BU difference-list method
Use of append/3</td><td>BU difference-list method
Use of explicit unification
(explicit cons operation)</td><td>BU difference-list method
Use of cons/2 procedure
(implicit cons operation)</td></tr>
<tr><td><pre>copy_bu_ap([],L,L).
copy_bu_ap([X|Xs],A,R)
:- append([X],A,Z),
   copy_bu_ap(Xs,Z,R).</pre></td><td><pre>d_copy_bu1([],L-L).
d_copy_bu1([X|Xs],R-A)
:- Z=[X|A],
   d_copy_bu1(Xs,R-Z).</pre></td><td><pre>d_copy_bu2([],L-L).
d_copy_bu2([X|Xs],R-A)
:- cons(X,Z-A),%Z=[X|A]
   d_copy_bu2(Xs,R-Z).</pre></td></tr>
</table>

Fig. 13. Definitions for BU-copying a list using implicit and explicit cons operations

<table>
<tr><th>Top-Down construction of lists</th><th>Bottom-Up construction of lists</th></tr>
<tr><td><pre>copy_td_ap([],L,L).
copy_td_ap([X|Xs],A,R):-
  copy_td_ap(Xs,A,RR),
  <b>append</b>([X],RR,R).
?- copy_td_ap([2,1,3,4],[],X).
X = [2, 1, 3, 4]</pre></td><td><pre>copy_bu_ap([],L,L).
copy_bu_ap([X|Xs],A,R) :-
   <b>append</b>([X],A,RR),
   copy_bu_ap(Xs,RR,R).
?- copy_bu_ap([2,1,3,4],[],X).
X = [4, 3, 1, 2]</pre></td></tr>
<tr><td><pre>dla_copy_td([],L-L).
dla_copy_td([X|Xs],   R-A):-
  dla_copy_td(Xs,   RR-A),
  <b>dl_append</b>([X|RR]-RR,RR-Z,R-Z).
?- dla_copy_td([2,1,3,4],X-[]).
X = [2, 1, 3, 4]</pre></td><td><pre>dla_copy_bu([],L-L).
dla_copy_bu([X|Xs],   R-A):-
  <b>dl_append</b>([X|A]-A,A-Z,RR-Z),
  dla_copy_bu(Xs,   R-RR).
?- dla_copy_bu([2,1,3,4],X-[]).
X = [4, 3, 1, 2]</pre></td></tr>
<tr><td><pre>da_copy_td([],L-L).
da_copy_td([X|Xs],   R-A):-
  da_copy_td(Xs,   RR-A),
  <b>dl_a</b>([X],   R-RR).
?- da_copy_td([2,1,3,4],X-[]).
X = [2, 1, 3, 4]</pre></td><td><pre>da_copy_bu([],L-L).
da_copy_bu([X|Xs],   R-A):-
  <b>dl_a</b>([X],   RR-A),
  da_copy_bu(Xs,   R-RR).
?- da_copy_bu([2,1,3,4],X-[]).
X = [4, 3, 1, 2]</pre></td></tr>
<tr><td><pre>d_copy_td([],L-L).
d_copy_td([X|Xs],R-A):-   % R-A
  d_copy_td(Xs,RR-A),   %+RR-A
  <b>cons</b>(X,R-RR).%≡R=[X|RR] %=R-RR
?- d_copy_td([2,1,3,4],X-[]).
X = [2, 1, 3, 4]</pre></td><td><pre>d_copy_bu([],L-L).
d_copy_bu([X|Xs],R-A):-   %R-A
  <b>cons</b>(X,RR-A),%RR=[X|A] %=RR-A
  d_copy_bu(Xs,R-RR).   %+R-RR
?- d_copy_bu([2,1,3,4],X-[]).
X = [4, 3, 1, 2]</pre></td></tr>
</table>

Fig. 14. Copying of lists using append/3 and its difference-list versions

above rule and compute the result R-A of the procedure `d_copy_bu2/2` from the difference-lists R-Z and Z-A (right-hand side in Figure 13).

The different programming methods are compared in Figure 14 applied to the problem of copying lists for both top-down (TD) and bottom-up (BU) generation of the result lists. The application of dl_append/3 demonstrates that an interpretation as formulated with Rule 1 is not possible. Moreover the structure of the arguments of the dl_append/3 call is rather complex. If a call of dl_a/2 instead of dl_append/3 is used, an interpretation in the manner of Rule 1 is possible, but dl_a/2 copies the list elements of its first argument, like append/3. Only where cons/2 is applied there is no copying of list elements and an interpretation corresponding to Rule 1 is possible. The

Table 3. Different approaches for TD/BU construction of lists

	dl_a/2	**dl_append/3**	**[X|L]**	**cons/2**
Property/ Functionality	Like append/3	Backtrack-free list concatenation	Explicit cons oper. append([E],X,Y) ≡ Y=[E|X]	Implicit cons operation
Semantics	Standard	Other semantics than append/3	Other semantics than append/3	Other semantics than append/3
Backtracking	Backtrackable	Not backtrackable	Not backtrackable	Not backtrackable
Typical application	dl_a([X],L-A)	dl_append([X|A]-A, A-Z, RR-Z)	dl_copy_bu(Xs, R - [X|A])	cons(X, Z-A)
Advantages	Supports programming with DL	No copying of list elements	No copying of list elements	Supports DL programming. No copying of list elements
Disadvantages	Copying of list elements. Infinite backtracking possible.	Does not support DL Programming	Does not support DL Programming	Self-defined procedure

Quicksort		
	Definitions using difference-lists	
Definitions using an append/3 call	Use of a call to cons/2	cons/2 unfolded (cons(H,L1-L2)≡L1=[H|L2])

`qsort([],[]).`
`qsort([X|Xs],R) :-`
`  split(X,Xs,A,B),`
`  qsort(A,T),`
`  qsort(B,Z),`
`  `**`append(T,[X|Z],R).`**
 	
`?- qsort([1,2],X).`
`X = [1,2]`
	`d_qsort([],L-L).`
	`d_qsort([X|Xs],R-RN):-`
	`  split(X,Xs,A,B),`
	`  d_qsort(A,R-R1),`
	`  d_qsort(B,R2-RN),`
	`  `**`cons(X,R1-R2).`**
	
	`?-d_qsort([1,2],X-[]).`
	`X = [1,2]`
		`du_qsort([],L-L).`
		`du_qsort([X|Xs],R-RN):-`
		`  split(X,Xs,A,B),`
		`  du_qsort(A,R-`**`[X|R2]`**`),`
		`  du_qsort(B,R2-RN).`
		
		`?-du_qsort([1,2],X-[]).`
		`X = [1,2]`

`qsort5([],[]).`
`qsort5([X|Xs],R) :-`
`  split(X,Xs,A,B),`
`  qsort5(B,Z),`
`  `**`append(T,[X|Z],R),`**
`  qsort5(A,T).`
`?-qsort5([1,2],X).`
`X = [1,2]`
`More?; Stack overflow`
	`d_qsort5([],L-L).`
	`d_qsort5([X|Xs],R-RN):-`
	`  split(X,Xs,A,B),`
	`  d_qsort5(B,R2-RN),`
	`  `**`cons(X,R1-R2),`**
	`  d_qsort5(A,R-R1).`
	`?-d_qsort5([1,2],X-[]).`
	`X = [1,2]`
		`du_qsort5([],L-L).`
		`du_qsort5([X|Xs],R-RN):-`
		`  split(X,Xs,A,B),`
		`  du_qsort5(B,R2-RN),`
		`  du_qsort5(A,L-`**`[X|R2]`**`).`
		
		`?-du_qsort5([1,2],X-[]).`
		`X = [1,2]`

Fig. 15. Proof of different semantics for procedures using append/3 and cons/2 calls

version d_copy_td/2 with TD construction of lists suffers from being non-rest-recursive. The reason is an in-place translation of the append/3 call of the original procedure, but in contrast to the append/3 call, the cons/2 call may be moved to another position without possibly changing the semantics of the procedure.

An overview of the results of the analysis of different approaches for TD and BU construction of lists is given in Table 3. Only the use of the newly introduced cons/2 procedure satisfies both demands for avoiding copying of lists and supporting DL-programming by allowing an interpretation according to Rule 1. The use of the cons/2 procedure results in more declarative programs, because these procedure calls may be moved freely around in the clause body. A similar property is not normally valid for append/3 calls. A call to append/3 may loop forever if the first and second arguments of append/3 are variables. This means that an automatic or manual transformation of a procedure that uses append/3 calls into an append-free procedure that uses calls to cons/2 may result in changed semantics (see also Figure 15).

For this reason and given the fact that we are able to propose a simple kind of DL programming, we believe that naive programming using append/3 calls and a subsequent manual or automatic transformation procedure are avoidable.

5 Rules for Constructing DL Procedures

5.1 Abstract Rules

The analysis of the different approaches to avoid copying lists and to support difference-list programming led to the introduction of a new procedure cons/2, which performs an implicit cons operation. A comparison of the application of cons/2 calls in procedures for TD and BU generation of lists is given in Figure 16. The only difference between the two procedures lies in the different "directions" of generation of the result list. The meaning of "top-down" (TD) may be: the first element (or first part) of the input list is taken to build the "top" difference-list L-L1, for "bottom-up" (BU), the same first part is taken to build the "bottom" difference-list L_{n-1}-Ln.

Properties	**Top-down construction of result list**	**Bottom-up construction of result list**
Base case	`d_copy_td2([],L-L).`	`d_copy_bu2([],L-L).`
Step case	`d_copy_td2([X|Xs],R-Rn):-`	`d_copy_bu2([X|Xs],R-Rn):-`
Splitting the	`cons(X,         R-R1),`	`cons(X,         R1-Rn),`
result list	`d_copy_td2(Xs, R1-Rn).`	`d_copy_bu2(Xs,  R-R1).`

Fig. 16. TD- and BU-construction of list using the cons/2 procedure

Figure 17 presents an abstract guideline for building TD and BU difference-list procedures. Note that the mentioned difference between the rules for TD and BU generation of result lists is the assignment of the parts of the resulting difference-list to the calls dealing with the elements of the input list (fourth item in the table for each case). The term "append-order" denotes the result list with the same order of the elements as in the input list (or more specifically: the same order in which the elements are chosen from the input list).

5.2 Unfolding Cons/2 Calls

Difference-list specification of Quicksort: The sorted difference-list R-RN for an input list [X|Xs] results from the composition of the difference-list for sorting all elements that are smaller than or equal to the current element and the sorted difference-list for all elements that are greater than the current element. The current element fits in the result list between the elements that are greater than and less than the current element.

In the top-down approach for construction of the result, the complete difference-list R-RN is composed from the difference-list R1-R2 for sorting the list elements that are smaller than X, from the difference-list for the element X, and from the difference-list R2-RN for sorting the elements of the input list that are greater than X (Figure 18).

	Structure of procedures in difference-list notation
Result with append-order of elements (TD construction of lists)	**Rules for a top-down construction of result lists** • Base case: The result parameter in the head of the clause (empty input list, empty result list) is the difference-list L-L. • Step case: The resulting difference-list in the head of the clause for the general case is R-R_n. • Splitting the result list: The result list could be split into difference-lists R-R_1, R_1- …- R_{n-1}, R_{n-1}-L_n, where R_1-…-R_{n-1} denotes an arbitrary set of further difference-lists. • TD assignment: The difference-list R-R_1 is used for dealing with the first selected element (or first part) of the input list, R_1-R_2 for dealing with the next element(s) of the input list, etc. • Unfolding: A cons/2 call may be used for dealing with a single element of the input list (unfolding the call of cons/2 supplies a more dense procedure definition; cons(X,R1-A) ≡ R1=[X|A]).
Result with inverse order of elements (BU construction of lists)	**Rules for a bottom-up construction of result lists** • Base case: The result parameter in the head of the clause (empty input list, empty result list) is the difference-list L-L. • Step case: The resulting difference-list in the head of the clause for the general case is R-R_n. • Splitting the result list: The result list could be split into difference-lists R-R_1, R_1- …- R_{n-1}, R_{n-1}-R_n, where R_1-…-R_{n-1} denotes an arbitrary set of further difference-lists. • BU assignment: The difference-list R_{n-1}-R_n is used for dealing with the first selected element (or first part) of the input list, R_{n-2}-R_{n-1} for dealing with the next element(s), etc. • Unfolding: A cons/2 call may be used for dealing with a single element of the input list (unfolding the call of cons/2 supplies a more dense procedure definition; cons(X,R1-A) ≡ R1=[X|A])

Fig. 17. Rules for difference-list notation of procedures

<table>
<tr><th colspan="2">Quick-Sort, TD construction of lists</th></tr>
<tr><td colspan="2">append order of the elements in the result list compared with input list</td></tr>
<tr><td>Use of a call of cons/2</td><td>Unfolding the call of cons/2</td></tr>
<tr><td>R-RN composed from R-R1, R1-R2, and R1-RN; result list starts with R-R1</td><td>Replacement of R1 by [X|R2];
crossing the cons/2-call</td></tr>
<tr><td>d_qsort_td([],L-L).
d_qsort_td([X|Xs], **R-RN**):-
  split(X,Xs,A,B),
  d_qsort_td(A, **R-R1**),
  /*L1=[X|L2]*/cons(X,**R1-R2**),
  d_qsort_td(B, **R2-RN**).
?- d_qsort_td([2,1,4,3], Res).
Res = [1, 2, 3, 4|L] - L</td><td>dl_qsort_td([],L-L).
dl_qsort_td([X|Xs],R-RN) :-
  split(X,Xs,A,B),
  dl_qsort_td(A, R-[X|R2]),
  dl_qsort_td(B, R2-RN).

?-dl_qsort_td([2,1,4,3],Res).
Res = [1, 2, 3, 4|L] - L</td></tr>
</table>

Fig. 18. TD difference-list construction for Quicksort

<table>
<tr><th colspan="2">Quick-Sort, BU construction of lists</th></tr>
<tr><td colspan="2">Inverse order of the elements in the result list compared with input list</td></tr>
<tr><td>Use of the cons/2 procedure</td><td>Unfolding the call of cons/2</td></tr>
<tr><td>R-RN composed from R2-RN, R1-R2, and R-R1; Result list ends with R-R1</td><td>Replacement of R1 by [X | R2];
crossing the cons/2 call</td></tr>
<tr><td>d_qsort_bu([],L-L).
d_qsort_bu([X|Xs], **R-RN**):-
  split(X,Xs,A,B),
  d_qsort_bu(A, **R2-RN**),
  /*L1=[X|L2]*/cons(X,**R1-R2**),
  d_qsort_bu(B, **R-R1**).
?-d_qsort_bu([2,1,4,3],Res).
Res = [4, 3, 2, 1|L] - L</td><td>dl_qsort_bu([],L-L).
dl_qsort_bu([X|Xs],R-RN):-
  split(X,Xs,A,B),
  dl_qsort_bu(A, R2-RN),
  dl_qsort_bu(B, R-[X|R2]).

?-dl_qsort_bu([2,1,4,3],Res).
Res = [4, 3, 2, 1|L] – L</td></tr>
</table>

Fig. 19. BU difference-list construction for quicksort

In the BU method for difference-list construction, we can either start with the first part of the input list together with the last part of the difference-list for the result, or we can start with the last part of the input list together with the initial part of the difference-list for the result. The corresponding procedures are called d_qsort_bu/2

<table>
<tr><th colspan="2">Quick-Sort, BU construction of lists</th></tr>
<tr><td colspan="2">Inverse order of the elements in the result list compared with input list</td></tr>
<tr><td>Use of the cons/2 operation</td><td>Unfolding the call of cons/2</td></tr>
<tr><td>d_qsort_bu2([],L-L).
d_qsort_bu2([X|Xs], **R-RN**):-
  split(X,Xs,A,B),
  d_qsort_bu2(B, **R-R1**),
  /*L1=[X|L2]*/cons(X,**R1-R2**),
  d_qsort_bu2(A, **R2-RN**).
?-d_qsort_bu2([2,1,4,3],Res)
Res = [4, 3, 2, 1|L] - L</td><td>dl_qsort_bu2([],L-L).
dl_qsort_bu2([X|Xs], R-RN) :-
  split(X,Xs,A,B),
  dl_qsort_bu2(B, R-[X|R2]),
  dl_qsort_bu2(A, R2-RN).

?-dl_qsort_bu([2,1,4,3],Res)
Res = [4, 3, 2, 1|L] – L</td></tr>
</table>

Fig. 20. Exchanged order of calls compared with Figure 19

(Figure 19) and d_qsort_bu2/2 (Figure 20). The versions with the unfolded call of cons/2 are called dl_qsort_bu/2 and dl_qsort_bu2/2.

The presented rule for difference-list programming and the tight connection between difference-list procedures and both top-down and bottom-up construction of lists could considerably facilitate the application of this technique. In several papers and textbooks, the connection between the difference-list technique and the accumulator technique is mentioned, but the complete relationship and its simplicity have not yet been described. In [1], for example , the authors, who deal with automatic transformation of lists into difference-list representation for a functional language, mention as differences between the dl technique and the accumulator technique the different positions of arguments and the different goal order (first and second column in Figure 21). However, the different goal order is not necessary and is therefore no feature of a transformation from an accumulator representation into a d representation. On the contrary, because of possible side effects or infinite loops in the called procedures, an exchange of calls, as performed in the procedure acc_qs/3, is normally not possible.

Quicksort - Comparison of dl and accumulator representation		
Equivalence described in [AFSV02]		Accumulator procedure analogous to du_qs/2
dl procedure (with unfolded cons/2 call)	Accumulator procedure with exchanged call order	

`du_qs([],L-L).`
`du_qs([X|Xs],R-RN):-`
`  split(X,Xs,A,B),`
`  du_qs(A,R-[X|R1]),`
`  du_qs(B,R1-RN).`
`?-du_qs([2,1],X-[]).`
`X = [1,2]`
	`acc_qs([],L,L).`
	`acc_qs([X|Xs],RN,R):-`
	`  split(X,Xs,A,B),`
	`  acc_qs(B,RN,R1),`
	`  acc_qs(A,[X|R1],R).`
	`?-acc_qs([2,1],X-[]).`
	`X = [1,2]`
		`acc_qsort([],L,L).`
		`acc_qsort([X|Xs],RN,R):-`
		`  split(X,Xs,A,B),`
		`  acc_qsort(A,[X|R1],R),`
		`  acc_qsort(B,RN,R1).`
		`?-acc_qsort([2,1],[],X).`
		`X = [1,2]`

Fig. 21. Comparison of quicksort in difference-list and accumulator representation (du_qs/2 and acc_qs/2 are syntactic variants of the procedures given in [AFSV02])

5.3 TD /BU Procedures

The almost identical schemes for top-down and bottom-up construction of lists allow us to transform an input into a list with elements and a list with the same elements in inverse order in one procedure at the same time. To avoid doubling the calls to cons/2, an extended procedure cons/3 is introduced, which puts the selected element X into two difference-lists.:

```
cons(X, [X|Xs]-Xs, [X|Ys]-Ys).                (8)
```

Figure 22 summarizes the construction rules for generation of result lists in append order and inverse order in one procedure at the same time.

An example of a TD/BU procedure is given in Figure 23. If only one of these result lists is needed, a possible disadvantage of such a TD/BU procedure would be the additional time needed to generate the second difference-list.

Rules for TD/BU construction of lists

- **Base case:** The two result parameters in the clause for the base case (empty input list, empty result list) are two difference-lists R-R and S-S, where R and S are both different from each other and from other variables in the clause.
- **Step case:** The result parameters in the head of the clause for the general case are R-R_n and S-Sn, where R, R_n, S and S_n are both different from each other and from other variables in the clause. R-R_n is a difference-list whose denotation is a list with elements in append order, S-S_n is a difference-list whose denotation is a list in inverse order.
- **Splitting the result parameters:** The resulting difference-lists R-R_n and S-S_n in the body of the clause for the general case could be split into difference-lists R-R_1, R_1- …- R_{n-1}, R_{n-1}-R_n, and S-S_1, S_1-…-S_{n-1}, S_{n-1}-S_n, where R_1-…-R_{n-1} and S_1-…-S_{n-1} denote arbitrary sets of further difference-lists.
- **TD/BU assignment:** The difference-lists R-R_1 and S_{n-1}-S_n are used for dealing with the first selected element (or first part) of the input list, and R_1 - R_2 and (S_{n-2})-(S_{n-1}) for dealing with the next element(s) of the input list, etc.
- **Unfolding:** The cons/3 operation may be used for dealing with a single element of the input list (unfolding the call of cons/3 supplies a more dense procedure definition; cons(X,L1-A1,L2-A2) ≡ L1=[X|A1] and L2=[X|A2]).

Fig. 22. Rule for construction of result lists in append order and reverse order in one procedure at the same time

TD/BU quicksort

```
%qsort(Input, List_in_append_order, List_in_reverse_order)

d_qsort([],R-R, S-S).
d_qsort([X|Xs],         R-RN,     S-SN) :-
   split(X,Xs,A,B),
   d_qsort(A,           R-R1,    S2-SN),
   cons(X,             R1-R2,    S1-S2),
   d_qsort(B,          R2-RN,     S-S1).
```

Fig. 23. Quicksort with result lists in append and reverse order

6 Summary and Future Work

We have considered ways of dealing with difference-lists in the literature and have investigated simple examples of constructing result lists from input lists. We are interested in a common mechanism to obtain the result list of a list-processing procedure in either a top-down or bottom-up manner. For efficiency reasons such a mechanism must avoid calls to append/3, avoid left-recursive programs and should avoid unnecessary backtracking steps.

The advantages of using the difference-list techniques for efficiency of programs have been stressed by many authors. Several textbooks contain examples of the application of difference-lists. Some authors refer to a tight connection between the accumulator technique and the difference-list technique. However, what is still lacking is a simple guideline on how to program with difference-lists. The matter has remained difficult. There have been some attempts to transform programs using calls of append/3 to concatenate lists or to insert elements into lists by automatic program transformation into difference-list programs.

From all this information, we have derived simple rules for manual programming with difference-lists. These rules were verified using various examples. We believe that it is much simpler to program directly with difference-lists than to use program transformation to obtain a difference-list program. The programmer is able to maintain and debug his/her original program more easily because he/she understands it better than a transformed program. Program transformation must preserve the semantics of the original program. If the transformation engine cannot decide whether the variable L in [X|L] denotes the empty list or a variable that does not occur elsewhere, a transformation into the difference-list notation must not be performed.

Difference-list procedures are more declarative and robust than procedures that use calls of append/3, where an infinite loop may occur if, say, the first and third arguments are variables. Avoiding calls of append/3 would avoid such errors as it could be demonstrated with the quicksort procedure (see also Figure 15). There are six possible permutations of two recursive calls and the call to append/3 in the clause for the step case. For one of these permutations, the quicksort procedure loops infinitely but the corresponding difference-list procedure does not.

The main advantage, from the programmer´s point of view, is that we have provided simple, schematic rules for using difference-lists. Our rules generalize both bottom-up construction of lists using accumulators and top-down construction of lists using calls to append/3 to the notion of difference-list. The introduction of the cons/2 procedure serves as a didactic means to facilitate and simplify the use of difference-lists. Calls of cons/2 could easily be removed from the procedures by an unfolding operation.

In our work so far, our interst has focused exclusively on the definition of the rules supporting programming with difference-lists. Future work should deal with a detailed analysis of the effects in terms of time and memory consumption. We have already noticed very different speedup rates for difference-list versions of procedures compared with the original procedures. There also seem to be significant differences in implementations of Prolog systems.

References

1. Albert, E., Ferri, C., Steiner, F., Vidal, G.: Improving functional logic programs by difference-lists. In: He, J., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 237–254. Springer, Heidelberg (2000)
2. Colhoe, H., Cotta, J.C.: Prolog by Example. Springer, Heidelberg (1988)
3. Clocksin, W.F.: Clause and Effect. Prolog Programming for the Working Programmer. Springer, Heidelberg (1997)
4. Clocksin, W.F.: Prolog Programming. Lecture Prolog for Artificial Intelligence 2001-02 at University of Cambridge, Computer Laboratory (2002)
5. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer, Heidelberg (1981, 1984, 1987)

6. Clark, K.L., McCabe, F.G.: Micro-Prolog: Programming in Logic. Prentice Hall International, Englewood Cliffs (1984)
7. Clark, K.L., Tärnlund, S.Å.: A First Order Theory of Data and Programs. In: Gilchrist, B. (ed.) Information Processing, pp. 939–944. North Holland, Amsterdam (1977)
8. Dodd, T.: Prolog. A Logical Approach. Oxford University Press, Oxford (1990)
9. Geske, U.: How to teach difference-lists. Tutorial. In: Online Proceedings – WLP 2006 (2006) (last visited, September 06, 2007), `http://www.kr.tuwien.ac.at/wlp06/T02-final.ps.gz`
10. Marriott, K., Søndergaard, H.: Prolog Transformation by Introduction of Difference-Lists. TR 88/14. Dept. CS, The University of Melbourne (1988)
11. Marriott, K., Søndergaard, H.: Prolog Difference-list transformation for Prolog. New Generation Computing 11, 125–157 (1993)
12. Maier, D., Warren, D.S.: Computing with Logic. The Benjamin/Cummings Publisher Company, Inc. (1988)
13. O'Keefe, R.A.: The Craft of Prolog. The MIT Press, Cambridge (1990)
14. Sterling, L., Shapiro, E.: The Art of Prolog. MIT Press, Cambridge (1986); seventh printing (1991)
15. Zhang, J., Grant, P.W.: An automatic difference-list transformation algorithm for Prolog. In: Kodratoff, Y. (ed.) Proc. 1988 European Conf. Artificial Intelligence, pp. 320–325. Pittman (1988)

Linear Weighted-Task-Sum – Scheduling Prioritized Tasks on a Single Resource*

Armin Wolf[1] and Gunnar Schrader[2]

[1] Fraunhofer FIRST, Kekuléstr. 7, D-12489 Berlin, Germany
Armin.Wolf@first.fraunhofer.de
[2] sd&m AG, Kurfürstendamm 22, D-10719 Berlin, Germany
gunnar.schrader@sdm.de

Abstract. Optimized task scheduling is in general an NP-hard problem, even if the tasks are prioritized like surgeries in hospitals. Better pruning algorithms for the constraints within such constraint optimization problems, in particular for the constraints representing the objectives to be optimized, will result in faster convergence of branch & bound algorithms.

This paper presents new pruning rules for linear weighted (task) sums where the summands are the start times of tasks to be scheduled on an exclusively available resource and weighted by the tasks' priorities. The presented pruning rules are proven to be correct and the speed-up of the optimization is shown in comparison with well-known general-purpose pruning rules for weighted sums.

1 Motivating Introduction

The allocation of non-preemptive activities (or tasks) on exclusively allocatable resources occurs in a variety of application domains: job-shop scheduling, time tabling as well as in surgery scheduling. Very often, the activities to be scheduled have priorities that have to be respected.

Example 1. Typically, the sequence of surgeries is organized with respect to the patients' *ASA scores.* These values, ranging from 1 to 5, subjectively categorize patients into five subgroups by preoperative physical fitness and are named after the *American Association of Anaesthetists* (ASA). In the clinical practice, it is very common to schedule high risk patients, i.e. with a high ASA score, as early as possible and short surgeries before the long ones: a typical schedule is shown in Figure 1.

In the given example it is easy to respect the priorities of the activities, i.e. the ASA scores of the surgeries and their durations: it is only necessary to establish

* The work presented in this paper is funded by the European Union (EFRE) and the state of Berlin within the framework of the research project "inubit MRP", grant no. 10023515.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 21–37, 2009.

Operating Room # 1			Tuesday, 15th November 2005			
No.	Patient Code	Surgery	Start Time	End Time	ASA score	Duration
1	3821	gastric banding	7:30 h	8:00 h	3	0:30
2	5751	appendectomy	8:00 h	8:30 h	3	0:30
3	2880	inguinal hernia, left	8:30 h	9:15 h	3	0:45
4	3978	fundoplication	9:15 h	10:45 h	3	1:30
5	7730	appendectomy	10:45 h	11:15 h	2	0:30
6	3881	gastric banding	11:15 h	11:55 h	2	0:40
7	3894	inguinal hernia, left	11:55 h	12:40 h	2	0:45
8	7962	fundoplication	12:40 h	15:10 h	2	2:30
9	8263	inguinal hernia, right	15:10 h	15:40 h	1	0:30
10	8120	inguinal hernia, right	15:40 h	16:25 h	1	0:45
11	8393	umbilical hernia	16:25 h	17:25 h	1	1:00
12	3939	enterectomy	17:25 h	19:25 h	1	2:00

Fig. 1. A typical surgery schedule respecting the ASA scores

some ordering constraints stating that all surgeries with ASA score 3 are scheduled before all surgeries with value 2, which have to be scheduled before the ones with value 1. Further, these ordering constraints have to state that within the surgeries having the same ASA scores the shorter have to be before the longer ones. However, real life is more complicated: sometimes it is better to schedule a patient with low risk and a simple surgery between two complicated surgeries of high or moderate risk. Furthermore, beyond the surgeries' ASA scores and their durations other objectives like the equipment to be used, the operation team members or the sepsis risk often have to be optimized, too. In these cases, the presented optimization criteria cannot be reflected by any "hard" constraints, because otherwise there might be a contradiction with other possibly controversial optimization criteria. Therefore, we propose to represent a prioritization of patients with high ASA scores and long surgery durations by a sum of the surgeries' start times weighted by factors depending on their patients' ASA scores. This sum might be one factor in a combining objective function, which is also influenced by other optimization criteria.

Concerning Example 1, the schedule presented in Figure 1 is intuitively perfect but mathematically suboptimal if we minimize the sum of start times of the surgeries weighted by their ASA scores 3, 2, 1: the weighted sum's value is 5305 minutes if every start time is coded as the minutes after 7:30 a.m. An optimal but counterintuitive schedule where the surgeries are weighted by their ASA scores is presented in Figure 2, i.e. the surgeries are not ordered with respect to their ASA scores. In this schedule the sum value of the surgeries' weighted start times is 4280 minutes. However, if we use the weights 10^{ASA}, i.e. 1000, 100, 10 instead of 3, 2, 1, the first schedule is intuitive and optimal – its sum value is 315300 minutes.

Optimized scheduling of tasks on exclusively available resources, e.g., where the objective is a sum depending on the tasks' start times, are in general

Operating Room # 1			Tuesday, 15th November 2005			
No.	Patient Code	Surgery	Start Time	End Time	ASA score	Duration
1	3821	gastric banding	7:30 h	8:00 h	3	0:30
2	5751	appendectomy	8:00 h	8:30 h	3	0:30
3	7730	appendectomy	8:30 h	9:00 h	2	0:30
4	2880	inguinal hernia, left	9:00 h	9:45 h	3	0:45
5	3881	gastric banding	9:45 h	10:25 h	2	0:40
6	3894	inguinal hernia, left	10:25 h	11:10 h	2	0:45
7	8263	inguinal hernia, right	11:10 h	11:40 h	1	0:30
8	3978	fundoplication	11:40 h	13:10 h	3	1:30
9	8120	inguinal hernia, right	13:10 h	13:55 h	1	0:45
10	8393	umbilical hernia	13:55 h	14:55 h	1	1:00
11	7962	fundoplication	14:55 h	17:25 h	2	2:30
12	3939	enterectomy	17:25 h	19:25 h	1	2:00

Fig. 2. A surgery schedule minimizing the sum of start times weighted by their ASA scores

NP-hard problems (cf. [2]). Nevertheless, constraint programming (CP) focuses on these problems, resulting in polynomial algorithms for pruning the search space (e.g., [2, 3, 5, 13]) as well as in heuristic and specialized search procedures for finding solutions (e.g., [11, 14, 15]).

Thus, obvious and straightforward approaches to solve such optimization problems in CP would be mainly based on two global constraints: a constraint stating that the tasks are not overlapping in time on the one hand and on the other hand on a sum of the tasks' weighted start times.

Example 2. Let three tasks be given which will be allocated to a common exclusively available resource:

- task no. 1 with duration $d_1 = 3$ and priority $\alpha_1 = 3$,
- task no. 2 with duration $d_2 = 6$ and priority $\alpha_2 = 3$,
- task no. 3 with duration $d_3 = 6$ and priority $\alpha_3 = 2$.

The tasks must not start before a given time 0, i.e. their not yet determined start times (variables) s_1, s_2 and s_3, respectively, must be non-negative. Furthermore the objective $r = \alpha_1 \cdot s_1 + \alpha_2 \cdot s_2 + \alpha_3 \cdot s_3$ has to be minimized. An obvious CP model of this problem in any established CP system over finite domains (e.g., CHIP, ECL^{i}PS^e, SICStus, etc.) is

$$\begin{aligned} &\mathsf{disjoint}([s_1, s_2, s_3], [3, 6, 6]) \\ &\wedge s_1 \geq 0 \wedge s_2 \geq 0 \wedge s_3 \geq 0 \wedge r = 3 \cdot s_1 + 3 \cdot s_2 + 2 \cdot s_3 \ , \end{aligned}$$

where the constraint $\mathsf{disjoint}([s_1, \ldots, s_n], [d_1, \ldots, d_n])$ states that either $s_i + d_i \leq s_j$ or $s_j + d_j \leq s_i$ holds for $1 \leq i < j \leq n$.[1] Any pruning of the variables' domains

[1] In some CP systems temporal non-overlapping of tasks in represented by the so-called $\mathsf{sequence}$ constraint instead of the $\mathsf{disjoint}$ constraint.

yields that the minimal value of r is 0. This pruning is rather weak. Now, if we consider all six possible sequences of the three tasks, i.e. $(1, 2, 3), \ldots, (3, 2, 1)$ a better lower bound of r is computable. For example, if we consider the task sequence $1, 2, 3$, the earliest start times are $s_1 = 0, s_2 = 3, s_3 = 9$ and thus $r = 3 \cdot 0 + 3 \cdot 3 + 2 \cdot 9 = 27$. However, if we consider the task sequence 3, 2, 1, the earliest start times are $s_1 = 12, s_2 = 6, s_3 = 0$ and thus $r = 3 \cdot 12 + 3 \cdot 6 + 2 \cdot 0 = 54$ which is not a lower bound of r. Considering all task sequences the minimal admissible value of r is 27 due to the temporal non-overlapping of the tasks. Unfortunately, this knowledge is not considered during the pruning of the variables' domains in the weighted sum.

Now, if we schedule task no. 2 to be first, i.e. $s_2 = 0$, then either the task sequence $2, 1, 3$ or $2, 3, 1$ is possible. In the first case, the earliest start times are $s_1 = 6, s_3 = 9$ and thus $r = 3 \cdot 6 + 3 \cdot 0 + 2 \cdot 9 = 36$. In the last case, the earliest start times are $s_1 = 12, s_3 = 6$ and thus $r = 3 \cdot 12 + 3 \cdot 0 + 2 \cdot 6 = 48$. It follows that the minimal admissible value of r is 36. This might contradict an upper bound of the variable (e.g., 35) set by a branch & bound optimization to minimize r, hence terminating the search for a better solution. Without this knowledge, the minimal value of r is 30 if we add $s_2 = 0$ to the CP model with the weighted sum constraint: In general the pruning algorithms for exclusively available resources (cf. [2, 10, 12, 13]) will prune the potential start times of the remaining tasks to be at least $6 = s_2 + d_2$. Thus, the earliest possible start times of the tasks no. 1 and no. 3 is 6, which computes to $\min(R) = 3 \cdot 6 + 3 \cdot 0 + 2 \cdot 6 = 30$. Thus, the detection of a contradiction with r's upper bound 35 requires the allocation of at least one more task. – Summarizing, we observe a well-known situation in CP modeling: "The drawback of these approaches comes from the fact that the constraints are handled independently" [7].

2 Aim, Related Work and Contribution

The aim of the presented work is an efficient calculation of a better bounds of any variable r constrained to $r = \sum_i^n \alpha_i s_i$ where $\alpha_i \in \mathbb{Q}$ are positive *weights* and s_i are the variable start times of some tasks that has to processed sequentially without temporal overlapping. "Better" means better than the lower bound that is computable in interval arithmetics for weighted sums in general. "Efficient" means without consideration of all possible task sequences – $O(n!)$ in the worst case.

The idea to consider constraints "more globally" while processing others is well known and well established (see e.g., [4]) because it results in better pruning of the search space and thus in better performance. Surprisingly, only a few proposals are made to combine the knowledge of so-called *global* constraints, too. In [1] an extension of the alldifferent constraint is made to consider inter-distance conditions in scheduling. In [7] a new constraint is proposed that combines a sum and a difference constraint. To our knowledge, in this article a proposal is made the first time that combines a disjoint constraint (cf. [2, 10, 12]) with a weighted sum (see e.g., [8]) and shows its practical relevance. In the following, we introduce new techniques for this new *global constraint* that allows us to tackle the fact that the variables in the summands of the considered weighted sums are the start times of tasks to be sequentially ordered, i.e. allocated to an exclusively available resource. This knowledge yields stronger and efficient pruning resulting in earlier detections of dead ends during the search for a minimal sum value. This is shown by experimental results.

The paper is organized as follows: next, the considered problems, i.e. *prioritized task scheduling problems* are formally defined. Second, the general pruning rules for weighted sums are recapitulated. Based on these rules, we deduce new ones for sums where the summands are weighted start times of tasks to be processed sequentially are presented. Additionally, we prove their correctness formally. Further, the rules' proper integration in constraint programming is described. Then, the better performance of the new pruning rules compared to the more general ones is shown by empirical examinations of different kinds of surgery optimization problems. Last but not least, the paper concludes with some remarks.

3 Definitions

Formally, a *prioritized task scheduling problem* like the scheduling of surgeries presented in Example 1 is a constraint optimization problem (COP) over finite domains. This COP is characterized by a finite task set $T = \{t_1, \ldots, t_n\}$. Each task $t_i \in T$ has an a-priori determined, positive integer-valued duration d_i. However, the start times $s_1, \ldots, s_n$ of the tasks in T are initially finite-domain variables having integer-valued finite potential start times, i.e. the constraints $s_i \in S_i$ hold for $i = 1, \ldots, n$, where S_i is the set of potential start times of the task $t_i \in T$. Furthermore, each task $t_i \in T$ has a fixed positive priority $\alpha_i \in \mathbb{Q}$.

Basically, the tasks in T have to be scheduled non-preemptively and sequentially – neither with any break while performing a task nor with any temporal overlap of any two tasks – on a commonly used exclusively available resource (e.g., a machine, an employee, etc. or especially an operating room). However, breaks between different tasks are allowed, e.g., for setting-up the used machine.

Ignoring other objectives and for simplicity's sake, the tasks in T have to be scheduled such that the sum of prioritized start times is minimal.[2] Thus, the first part of the *problem* is to find a *solution*, i.e. some start times $s_1 \in S_1, \ldots, s_n \in S_n$ such that $\bigwedge_{1 \leq i < j \leq n} (s_i + d_i \leq s_j \vee s_j + d_j \leq s_i)$ holds.[3] Furthermore, this solution has to be *minimal*, i.e. it minimizes the sum $\sum_{i=1}^{n} \alpha_i \cdot s_i$ such that for any other solution $s'_1 \in S_1, \ldots, s'_n \in S_n$ it holds: $\sum_{i=1}^{n} \alpha_i \cdot s_i \leq \sum_{i=1}^{n} \alpha_i \cdot s'_i$.

In the following, we call this problem, which is characterized by a task set T, the *Prioritized Task Scheduling Problem* of T, or PTSP(T) for short. Furthermore, we call a PTSP(T) *solvable* if a (minimal) solution exists.

[2] The maximization of the sum is a symmetric problem and handled analogously.

[3] The inequalities reflect the fact that breaks between tasks are allowed.

4 Better Pruning for the PTSP

In the following we consider a weighted sum constraint $r = \sum_{i=1}^{n} \alpha_i \cdot s_i$ with positive rational factors $\alpha_1, \ldots, \alpha_n$ and finite domain variables $s_1 \in S_1, \ldots, s_n \in S_n$ and $r \in R$. In detail, the domains of these variables are finite sets of integer values. A pruning scheme for such a weighted sum has been proposed in [8]. It works as follows. Based on the domains' minimal and maximal values new lower and upper bounds (lwbs and upbs) for the variables' values are computed:

$$\mathsf{lwb}(s_k) = \left\lceil \frac{1}{\alpha_k} \cdot \min(R) - \sum_{i=1, i \neq k}^{n} \frac{\alpha_i}{\alpha_k} \cdot \max(S_i) \right\rceil \quad \text{and}$$

$$\mathsf{upb}(s_k) = \left\lfloor \frac{1}{\alpha_k} \cdot \max(R) - \sum_{i=1, i \neq k}^{n} \frac{\alpha_i}{\alpha_k} \cdot \min(S_i) \right\rfloor$$

for $k = 1, \ldots, n$ and in particular

$$\mathsf{lwb}(r) = \left\lceil \sum_{i=1}^{n} \alpha_i \cdot \min(S_i) \right\rceil \quad \text{and} \quad \mathsf{upb}(r) = \left\lfloor \sum_{i=1}^{n} \alpha_i \cdot \max(S_i) \right\rfloor \ .$$

Then, the domains are updated with these bounds[4]:

$$S'_k = S_k \cap [\mathsf{lwb}(s_k), \mathsf{upb}(s_k)] \text{ for } k = 1, \ldots, n \text{ and } R' = R \cap [\mathsf{lwb}(r), \mathsf{upb}(r)] \ .$$

Finally, this process is iterated until a fix-point is reached, i.e. none of the domains change anymore.

As shown in Example 2, this pruning procedure applied to an PTSP(T) is too general if the minimal start times of all tasks have almost the same value. In this case, the fact that the variables $s_1, \ldots, s_n$ are start times of tasks to be serialized is not considered. Also shown in Example 2, this results in rather poor pruning and rather late detection of dead ends during a branch & bound search for good or even best solutions where the greatest value of r is bound to the actual objective value. We must therefore look for an alternative approximation of the lower bound of the variable r using the knowledge about the variables in the sum's summands.

Observation: Given a PTSP(T) with the task set $T = \{t_1, \ldots, t_n\}$ then for any solution $\tilde{s}_1 \in S_1, \ldots, \tilde{s}_n \in S_n$ such that $\tilde{s}_{\sigma(i)} + d_{\sigma(i)} \leq \tilde{s}_{\sigma(j)}$ is satisfied for $1 \leq i < j \leq n$ and a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ it holds

$$\begin{aligned} \tilde{s}_{\sigma(1)} + d_{\sigma(1)} \leq \tilde{s}_{\sigma(2)} & \\ \tilde{s}_{\sigma(2)} + d_{\sigma(2)} \leq \tilde{s}_{\sigma(3)} & \\ \vdots \quad \leq \ldots \leq \tilde{s}_{\sigma(n-1)} & \\ \tilde{s}_{\sigma(n-1)} + d_{\sigma(n-1)} \leq \tilde{s}_{\sigma(n)} & \end{aligned}$$

or more condensed: $\tilde{s}_{\sigma(1)} + \sum_{j=1}^{i-1} d_{\sigma(j)} \leq \tilde{s}_{\sigma(i)}$ for $i = 1, \ldots, n$. It follows that

[4] The updated domains are quoted for a better discrimination.

$$\sum_{i=1}^{n} \alpha_i \cdot \tilde{s}_i = \sum_{i=1}^{n} \alpha_{\sigma(i)} \cdot \tilde{s}_{\sigma(i)} \geq \sum_{i=1}^{n} \alpha_{\sigma(i)} \cdot \Big(\sum_{j=0}^{i-1} d_{\sigma(j)} \Big)$$

holds, if we define $\sigma(0) = 0$ and $d_0 \leq \min(S_{\sigma(1)})$ to be a lower bound of the earliest start time of $s_{\sigma(1)}$. This means that we are interested in a permutation $\theta : \{0, \ldots, n\} \to \{0, \ldots, n\}$ with $\theta(0) = 0$ and, e.g.; $d_0 = \min(S_1 \cup \ldots \cup S_n)$ such that

$$\sum_{i=1}^{n} \alpha_i \cdot s_i \geq \sum_{i=1}^{n} \alpha_{\theta(i)} \cdot \Big(\sum_{j=0}^{i-1} d_{\theta(j)} \Big)$$

holds for *all* solutions of the PTSP(T). This observation is the basis for better approximations of the bounds of r resulting from some theoretical examinations concerning the ordering on the tasks in T. Therefore, the following lemma formulates a crucial property using the fact that it is always possible to sort the tasks in T such that $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$.

Lemma 1. *Consider a PTSP(T) with the tasks in $T = \{t_1, \ldots, t_n\}$ be ordered such that*

$$\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j} \text{ holds for } 1 \leq i < j \leq n$$

and $d_0 = \min(S_1 \cup \ldots \cup S_n)$ be the earliest potential start time of all tasks in T.

Further consider a permutation $\sigma : \{0, 1, \ldots, n\} \to \{0, 1, \ldots, n\}$ with $\sigma(0) = 0$ and an index $k \in \{1, \ldots, n-1\}$ such that the inequality $\sigma(k) > \sigma(k+1)$ holds.

Then swapping the k-th and the $k+1$-th task never increases the total sum, i.e. it holds

$$\sum_{i=1}^{n} \alpha_{\theta(i)} \cdot \Big(\sum_{j=0}^{i-1} d_{\theta(j)} \Big) \leq \sum_{i=1}^{n} \alpha_{\sigma(i)} \cdot \Big(\sum_{j=0}^{i-1} d_{\sigma(j)} \Big)$$

for the resulting permutation θ with $\theta(i) = \sigma(i)$ for $i \in \{1, \ldots, n\} \setminus \{k, k+1\}$ and $\theta(k) = \sigma(k+1)$ as well as $\theta(k+1) = \sigma(k)$.

Proof. It holds that

$$\begin{aligned}\sum_{i=1}^{n}\alpha_{\theta(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\theta(j)}\Big) &= \sum_{i=1}^{k-1}\alpha_{\sigma(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\sigma(j)}\Big)+\alpha_{\sigma(k+1)}\cdot\Big(\sum_{j=0}^{k-1}d_{\sigma(j)}\Big)\\ &\quad+\alpha_{\sigma(k)}\cdot\Big(\sum_{j=0}^{k-1}d_{\sigma(j)}+d_{\sigma(k+1)}\Big)+\sum_{i=k+2}^{n}\alpha_{\sigma(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\sigma(j)}\Big)\\ &=\sum_{i=1}^{k-1}\alpha_{\sigma(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\sigma(j)}\Big)+\alpha_{\sigma(k+1)}\cdot\Big(\sum_{j=0}^{k}d_{\sigma(j)}\Big)\\ &\quad+\alpha_{\sigma(k)}\cdot d_{\sigma(k+1)}-\alpha_{\sigma(k+1)}\cdot d_{\sigma(k)}\\ &\quad+\alpha_{\sigma(k)}\cdot\Big(\sum_{j=0}^{k-1}d_{\sigma(j)}\Big)+\sum_{i=k+2}^{n}\alpha_{\sigma(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\sigma(j)}\Big)\\ &=\sum_{i=1}^{n}\alpha_{\sigma(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\sigma(j)}\Big)+\alpha_{\sigma(k)}\cdot d_{\sigma(k+1)}-\alpha_{\sigma(k+1)}\cdot d_{\sigma(k)}\end{aligned}$$

Further, with $\sigma(k+1)<\sigma(k)$ it also holds that $\alpha_{\sigma(k)}\cdot d_{\sigma(k+1)}\leq\alpha_{\sigma(k+1)}\cdot d_{\sigma(k)}$. Thus, the inequality

$$\sum_{i=1}^{n}\alpha_{\theta(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\theta(j)}\Big)\leq\sum_{i=1}^{n}\alpha_{\sigma(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\sigma(j)}\Big)$$

immediately follows: the summand $\alpha_{\sigma(k)}\cdot d_{\sigma(k+1)}-\alpha_{\sigma(k+1)}\cdot d_{\sigma(k)}$ has no positive value because of the assumed ordering of the tasks. □

The knowledge deduced in Lemma 1 is used in the following theorem to prove that the non-decreasing ordering of the tasks with respect to their duration/priority quotients results in a minimal sum of accumulated durations:

Theorem 1. *Let a PTSP(T) be given. Further, it is assumed that the tasks in* $T=\{t_1,\ldots,t_n\}$ *are ordered such that*

$$\frac{d_i}{\alpha_i}\leq\frac{d_j}{\alpha_j}\ \textit{holds for}\ 1\leq i<j\leq n.$$

Then, for an arbitrary permutation $\sigma:\{0,1,\ldots,n\}\rightarrow\{0,1,\ldots,n\}$ *with* $\sigma(0)=0$ *it holds*

$$\sum_{i=1}^{n}\alpha_i\cdot\Big(\sum_{j=0}^{i-1}d_j\Big)\leq\sum_{i=1}^{n}\alpha_{\sigma(i)}\cdot\Big(\sum_{j=0}^{i-1}d_{\sigma(j)}\Big)\ ,$$

where $d_0=\min(S_1\cup\ldots\cup S_n)$ *be the earliest potential start time of all tasks in* T*.*

Proof. The inequality to be shown is proven by induction over n, the numbers of tasks. Obviously, it holds for one task ($n = 1$): there is exactly one permutation σ – the identity.

For the induction step from n to $n+1$ let $n+1$ ordered tasks and an arbitrary permutation $\sigma : \{0, 1, \ldots, n, n+1\} \to \{0, 1, \ldots, n, n+1\}$ with $\sigma(0) = 0$ be given. Now during the induction step the fact is used that the permutation of the first n summands in the sum $\sum_{i=1}^{n+1} \alpha_{\sigma(i)} \cdot \left(\sum_{j=0}^{i-1} d_{\sigma(j)} \right)$ does not influence the value of the last summand $\alpha_{\sigma(n+1)} \cdot \left(\sum_{j=1}^{n} d_{\sigma(j)} \right)$. This results in the following two equalities and by induction in the final inequality:

$$\begin{aligned}
\sum_{i=1}^{n+1} \alpha_{\sigma(i)} \cdot \Big(\sum_{j=0}^{i-1} d_{\sigma(j)} \Big) &= \sum_{i=1}^{n} \alpha_{\sigma(i)} \cdot \Big(\sum_{j=0}^{i-1} d_{\sigma(j)} \Big) + \ \alpha_{\sigma(n+1)} \cdot \Big(\sum_{j=0}^{n} d_{\sigma(j)} \Big) \\
&= \sum_{i=1}^{n} \alpha_{\sigma(i)} \cdot \Big(\sum_{j=0}^{i-1} d_{\sigma(j)} \Big) + \ \alpha_{\sigma(n+1)} \cdot \Big(\sum_{j=0, j\neq\sigma(n+1)}^{n+1} d_j \Big) \\
&\geq \sum_{i=1, i\neq\sigma(n+1)}^{n+1} \alpha_i \cdot \Big(\sum_{j=0}^{i-1} d_j \Big) + \ \alpha_{\sigma(n+1)} \cdot \Big(\sum_{j=0, j\neq\sigma(n+1)}^{n+1} d_j \Big) \ .
\end{aligned}$$

Now, let $k = \sigma(n+1)$. If $k \neq n+1$, i.e. $k < n+1$ holds, the successive swaps of the k-th task and the $(n+1)$-th, …, $(k+1)$-th tasks yields to the sum $\sum_{i=1}^{n+1} \alpha_i \cdot \left(\sum_{j=0}^{i-1} d_j \right)$. Lemma 1 states that during these successive swaps the total sum is never increased. Concluding, the inequality to be proven holds:

$$\sum_{i=1}^{n+1} \alpha_{\sigma(i)} \cdot \Big(\sum_{j=0}^{i-1} d_{\sigma(j)} \Big) \geq \sum_{i=1}^{n+1} \alpha_i \cdot \Big(\sum_{j=0}^{i-1} d_j \Big) \ .$$

□

This means that Theorem 1 results in a non-trivial lower bound of r:

Corollary 1. *Let a PTSP(T) be given with $T = \{t_1, \ldots, t_n\}$ such that $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$. Further, let $d_0 = \min(S_1 \cup \ldots \cup S_n)$, the objective $r = \sum_{i=1}^{n} \alpha_i \cdot s_i$, and*

$$\mathsf{lwb}'(r) = \left\lceil \sum_{i=1}^{n} \alpha_i \cdot \Big(\sum_{j=0}^{i-1} d_j \Big) \right\rceil$$

be defined. Then, the pruning of the domain R of the sum variable r by updating $R' = R \cap [\mathsf{lwb}'(r), \mathsf{upb}(r)]$ is correct, i.e. all values of r which are part of any solution of the PTSP(T) are greater than or equal to $\mathsf{lwb}'(r)$. □

Analogously, there is a similar non-trivial approximation of the upper bound of the objective:[5]

[5] A formal proof of this symmetric statement is omitted because the argumentation is analogous.

Corollary 2. *Let a PTSP(T) be given with $T = \{t_1, \ldots, t_n\}$ such that $\frac{d_i}{\alpha_i} \geq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$. Further, let $e_n = \max(S_1 \cup \ldots \cup S_n)$, the objective $r = \sum_{i=1}^{n} \alpha_i \cdot s_i$, and*

$$\mathsf{upb'}(r) = \left\lfloor \sum_{i=1}^{n} \alpha_i \cdot (e_n - \sum_{j=i}^{n-1} d_j) \right\rfloor$$

be defined. Then, the pruning of the domain R of the sum variable r by updating $R' = R \cap [\mathsf{lwb'}(r), \mathsf{upb'}(r)]$ is correct, i.e. all values of r which are part of any solution of the PTSP(T) are especially less than or equal to $\mathsf{upb'}(r)$. □

5 Practical Application

For a practical application of Theorem 1 and its corollaries in a constraint programming system any change of the potential start times of the tasks has to be considered immediately. This means that the bounds of the variables have to be updated after any change of the variables domains. This update has to be performed in particular during the search for a solution where some of the tasks are already scheduled while others are not. These situations offer the opportunity to compute more adequate bounds for the weighted sum. This will be considered in the following.

Let a PTSP(T) with ordered tasks $T = \{t_1, \ldots, t_n\}$ be given such that $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$. Then, especially during the search for (minimal) solutions of the PTSP some of the tasks will be scheduled. Thus, there will be a partition of already scheduled tasks $S = \{t_{p_1}, \ldots, t_{p_q}\}$ and unscheduled tasks $U = \{t_{u_1}, \ldots, t_{u_v}\}$ such that $S \cup U = T$ and $S \cap U = \emptyset$ holds. For the scheduled tasks the start times are determined, i.e. $S_{p_1} = \{w_1\}, \ldots, S_{p_q} = \{w_q\}$. The start times of the unscheduled tasks are undetermined, i.e. $|S_{u_1}| > 1, \ldots, |S_{u_v}| > 1$.

Following Corollary 1 for all potential start times $s_1 \in S_1, \ldots s_n \in S_n$ it holds that

$$\begin{aligned}\sum_{i=1}^{n} \alpha_i \cdot s_i &= \sum_{i=1}^{q} \alpha_{p_i} \cdot s_{p_i} + \sum_{j=1}^{v} \alpha_{u_j} \cdot s_{u_j} \\ &\geq \sum_{i=1}^{q} \alpha_{p_i} \cdot w_i + \sum_{j=1}^{v} \alpha_{u_j} \cdot \Big(\sum_{k=0}^{j-1} d_{u_k}\Big)\end{aligned}$$

where $d_{u_0} = \min(S_{u_1} \cup \ldots \cup S_{u_v})$ is the earliest potential start time of all unscheduled tasks in U. This means that Corollary 1 results in another, more adequate, non-trivial lower bound of r:

$$\mathsf{lwb''}(r) = \left\lceil \sum_{i=1}^{q} \alpha_{p_i} \cdot w_i + \sum_{j=1}^{v} \alpha_{u_j} \cdot \Big(\sum_{k=0}^{j-1} d_{u_k}\Big) \right\rceil \quad .$$

Now, let the same PTSP(T) be considered. However, its tasks $T = \{t_1^*, \ldots, t_n^*\}$ are sorted in reverse order, i.e. $\frac{d_i^*}{\alpha_i^*} \geq \frac{d_j^*}{\alpha_j^*}$ holds for $1 \leq i < j \leq n$. Then, especially during the search for solutions of the PTSP some of the tasks will be scheduled. Again, there will be a partition of already scheduled tasks $S = \{t_{p_1}^*, \ldots, t_{p_q}^*\}$ and unscheduled tasks $U = \{t_{u_1}^*, \ldots, t_{u_v}^*\}$ such that $S \cup U = T$ and $S \cap U = \emptyset$ holds. For the scheduled tasks the start times are determined, i.e. $S_{p_1}^* = \{w_1^*\}, \ldots, S_{p_q}^* = \{w_q^*\}$. The start times of the unscheduled tasks are undetermined, i.e. $|S_{u_1}^*| > 1, \ldots, |S_{u_v}^*| > 1$.

Following Corollary 2 for all potential start times $s_1^* \in S_1^*, \ldots s_n^* \in S_n^*$ it holds that

$$\mathsf{upb}''(r) = \left\lfloor \sum_{i=1}^{q} \alpha_{p_i}^* \cdot w_i^* + \sum_{j=1}^{v} \alpha_{u_j}^* \cdot (e_U^* - \sum_{k=j}^{v-1} d_{u_k}^*) \right\rfloor .$$

where $e_U^* = \max(S_{u_1}^* \cup \ldots \cup S_{u_v}^*)$ is the latest potential start time of all unscheduled tasks in U.

Example 3 (Example 2 continued). Considering again the three tasks to be allocated to a common exclusively available resource. Now, if we schedule task no. 2 to be first with $s_2 = 0$, then in general the pruning algorithms (cf. e.g., [2, 10, 12, 13]) will prune the potential start times of the remaining tasks to be not less than $s_2 + d_2 = 6$. Thus, $d_{u_0} = \min(S_1 \cup S_3) \geq 6$ holds and it follows: $\alpha_1 \cdot s_1 + \alpha_2 \cdot s_2 + \alpha_3 \cdot s_3 \geq \alpha_2 \cdot s_2 + \alpha_1 \cdot d_{u_0} + \alpha_3 \cdot (d_{u_0} + d_1) \geq 3 \cdot 0 + 3 \cdot 6 + 2 \cdot (6+3) = 36$ – the value stated in Example 2 is now provided with evidence.

Corollary 3. *Let a PTSP(T) be given with $T = \{t_1, \ldots, t_n\}$ and the objective $r = \sum_{i=1}^{n} \alpha_i \cdot s_i$. Then, the pruning of the domain R of the sum variable r by updating*

$$R' = R \cap [\mathsf{lwb}''(r), \mathsf{upb}''(r)]$$

is correct, i.e. all values of r which are part of any solution of the PTSP(T) are greater than or equal to $\mathsf{lwb}''(r)$ *and less than or equal to* $\mathsf{upb}''(r)$. □

6 Empirical Examination

For an empirical examination of the influence of the presented bounds during the search for (minimal) solutions of PTSPs we implemented the pruning rules with the presented approximations lwb' and lwb" as well as their counterparts upb' and upb" in a specialized WeightedTaskSum constraint in our Java constraint solving library `firstCS` [6]. WeightedTaskSum specializes `firstCS`' WeightedSum constraints implementing the general pruning rules (cf. [8]) for weighted sums, i.e. $S = \sum_{i=1}^{n} \alpha_i A_i$ on finite domain variables $S, A_1, \ldots, A_n$. Additionally, the implementation of WeightedTaskSum takes into account that the variables $A_1, \ldots, A_n$ are the start times of tasks that must not overlap in time. Thus, the durations of the tasks are additional parameters of this constraint.

For the optimization we used a branch & bound approach based on a monotonic bounding scheme: given a (suboptimal) solution of the CSP of the considered COP with actual objective value obj an attempt is made to find a better solution with an objective value less than obj. The search continues until no better solution is found. Then, the most recently found solution is a minimal solution of the PTSP.

The used branching scheme is a left-to-right, depth-first incremental tree search which avoids re-traversals of already visited paths in the search tree containing suboptimal solutions (cf. [9]). The applied search strategy is specialized for contiguous task scheduling [14].

The pruning rules implemented in the WeightedTaskSum constraint are applied to real life task scheduling problems like the scheduling of surgeries in hospitals. The latter was indeed the ultimate cause to search for better and specialized pruning algorithms: the respond times of an interactive surgery scheduler based on our Java constraint solving library `firstCS` [6] were not satisfactory. With the help of the WeightedTaskSum constraint, we managed to solve a real world problem where the user expectations with regard to quality and speed could be fulfilled.

For the sake of simplicity, we will restrict ourselves to Example 1 in order to demonstrate how the WeightedTaskSum constraint can improve the search process. We compare the scheduling using the WeightedTaskSum and WeightedSum constraints, respectively. Both constraints were combined with an implementation of disjoint constraints, called SingleResource in `firstCS`. Any SingleResource constraint avoids overlapping of tasks on an exclusively available resource and performs *edge-finding*, *not-first-/not-last-detection* and other pruning strategies (cf. [13]). The results for the reunite comparison[6] are shown in Figure 3 and Figure 4.

The tables in both figures show how the runtime, the number of choices and backtracks increase by the increasing number of surgeries to be scheduled. In the first row the number of surgeries which have to be scheduled are indicated. For

Number of surgeries	PTSP(T) with WeightedTaskSum computation of a 1st opt. solution			PTSP(T) with WeightedSum computation of a 1st opt. solution		
	time/msec.	# choices	# backtracks	time/msec.	# choices	# backtracks
8	13	55	25	119	696	674
9	27	91	46	523	2842	2817
10	64	301	181	2212	12242	12177
11	135	623	396	13228	72035	71914
12	153	771	462	51198	302707	302552

Fig. 3. Comparison of the measured runtime, the choices made, and the performed backtracks for finding a minimal solution of a PTSP(T) using the WeightedTaskSum and WeightedSum constraints, respectively

[6] The results were obtained by using a Pentium IV PC with 3.06 GHz and 1 GB RAM.

Number of surgeries	PTSP(T) with WeightedTaskSum computation of a 1st opt. solution			PTSP(T) with WeightedSum computation of a 1st opt. solution		
	time/msec.	# choices	# backtracks	time/msec.	# choices	# backtracks
8	100	741	471	620	2177	2020
9	228	1690	1154	1857	7222	6934
10	343	2394	1629	6234	31637	31257
11	753	4908	3444	28640	155288	154610
12	1391	9050	6604	132000	717678	716669

Fig. 4. Comparison of the measured runtime, the choices made, and the performed backtracks for finding a minimal solution of a PTSP(T) using the reversed order on the tasks as well as the WeightedTaskSum and WeightedSum constraints, respectively

the purpose of comparison by the number of surgeries we shortened the schedule of Example 1 by the last surgeries of each ASA score category. This means, we considered the first surgery with ASA score 3, then the first surgery with ASA score 2, the first with score 1, the second with score 3, etc. in Figure 3 and the reversed order in Figure 4.

Obviously, the WeightedTaskSum constraint yields much faster to the first optimal solutions than the WeightedSum constraint. Whereas the runtime using the latter increases exponentially, the runtime using the WeightedTaskSum increases only slightly. During further examination of the search process we found out that the search using the WeightedSum constraint took up most of the time needed for finding the optimum before the proof of the optimum could be executed. However, both phases took much longer than the equivalent phases of the search using the WeightedTaskSum constraint.

At this point it has to be mentioned that the computed schedule – either using the WeightedTaskSum or the WeightedSum constraint – is the schedule presented in Figure 2 which is different from the intended schedule shown in Figure 1. This difference does not reflect an error. Rather, it is one of the cases – which we mentioned earlier in the introduction – where lower risk surgeries are scheduled before higher risk surgeries. However, if we want the exact ordering as shown in Figure 1 by using the weighted (task) sum constraints, we can increase the weights of the weighted (task) sum induced by the ASA scores non-proportionally: either as shown in the introductory Example 1 or by representing ASA score 3 by weight 300, 2 by 20 and 1 by itself. In any case an adequate representation – resulting in the desired ordering – can be computed easily: the weights have to be chosen such that $\frac{d_i}{\alpha_i} \leq \frac{d_j}{\alpha_j}$ holds for $1 \leq i < j \leq n$. When we increase the weights in the described manner the schedule will be computed not only exactly as shown in Figure 1.[7] In addition, the search which uses the WeightedSum constraint will find the first optimal solution much faster than before. Figure 5 shows how the increased weights (300 for 3 etc.) drastically change the comparison between

[7] The intended order is only computed if there are no additional constraints, e.g., time restrictions avoiding such a solution.

Number of surgeries	PTSP(T) with WeightedTaskSum computation of a 1st opt. solution			PTSP(T) with WeightedSum computation of a 1st opt. solution		
	time/msec.	# choices	# backtracks	time/msec.	# choices	# backtracks
8	24	78	30	40	95	51
9	28	115	42	67	150	93
10	67	208	90	136	279	185
11	132	324	135	261	421	276
12	149	438	174	283	548	328

Fig. 5. Comparison of the measured runtime, the choices made and the performed backtracks for finding a minimal solution with regard to the non-proportional increased weights and already chosen orders of surgeries

No. of surgeries	PTSP(T) with WeightedTaskSum computation of a 1st opt. solution			PTSP(T) with WeightedSum computation of a 1st opt. solution		
	time/msec.	# choices	# backtracks	time/msec.	# choices	# backtracks
8	292	976	421	462	986	461
9	738	1993	883	1175	2004	958
10	1796	4170	1913	3328	4450	2566
11	4406	8606	4113	7496	8942	5211
12	7649	13449	6613	13146	14550	8914

Fig. 6. Comparison of the measured runtime, the choices made and the performed backtracks for finding a minimal solution with regard to the non-proportional increased weights and reversed orders of surgeries

WeightedTaskSum and WeightedSum based search. Especially in the case where the surgeries are considered in reversed order – cf. Figure 6 – the runtime complexity of both optimizations behave similarly. However, the WeightedTaskSum based search still performs better than the WeightedSum based solution process.

Of course, the runtime depends not only on the total number of surgeries but also on the distribution of the ASA score and the duration. With the WeightedSum constraint the computation of the first optimal solution takes much longer when most (or all) surgeries have the same ASA score and/or the same duration. The reason for this lies in a weaker propagation which results in a much later detection of dead ends during the search.

In Figure 7 we demonstrate how the ASA scores and the durations having the same value each may effect the runtime. We compared the performance of the WeightedTaskSum together with SingleResource against the computation of the sums for all permutations of the considered surgeries, just generating all combinations. – The examination of such problems is not just an “academic” issue without any practical relevance. On the contrary, we were faced with these kinds of problems by user demands.

Number of surgeries	PTSP(T) with WeightedTaskSum computation of a 1st opt. solution			PTSP(T) with WeightedSum computation of **all** solutions	
n	time/msec.	# choices	# backtracks	time/msec.	# combinations $n!$
8	11	8	0	31	40320
9	15	9	0	313	362880
10	15	10	0	2648	3628800
11	17	11	0	27737	39916800
12	17	12	0	279418	479001600

Fig. 7. Comparison of the measured runtime, the made choices and the performed backtracks with the consideration of all combinations for finding a minimal solution with regard to problems where the ASA score and the duration have the same value each

The numbers in Figure 7 show that exponential exhaustive computations ($O(n!)$) are necessary if we use WeightedSum either together with SingleResource or any other, even specialized constraint for scheduling equal length tasks[8] (cf. [1]). However, WeightedTaskSum requires the least number of choices to find an optimal solution: n for the finding and proving the optimality where n is the number of surgeries.

7 Conclusion and Future Work

The consideration of prioritized task scheduling problems shows that specialized constraint problems allow specialized and in this case even stronger pruning rules. In this paper, we invented new pruning rules for weighted sums where the summands are the weighted start times of tasks to be serialized. Then, the presented theoretical results are adopted for their practical application in constraint programming. Further, their adequacy was practically shown in a real-world application domain: the optimal sequencing of surgeries in hospitals. There, in contrast to the original rules the new pruning rules behave in very robust way, especially in real scheduling problems.

The time complexity for optimization diversifies significantly while using the original pruning rules; they are more stable for the considered kinds of problems applying the new rules.

Future work will focus on a generalization of the presented results where the summands are more general functions of the start times. Even the case where the summands are weighted squares of the start times is encouraging because

$$(d_1)^2 + \cdots + (d_k)^2 \leq (d_1 + \cdots + d_k)^2$$

holds for any positive durations $d_1, \ldots, d_k$. This allows us to approximate the square of a start time s_{k+1} by

[8] Using SingleResource the elapsed times are even worse than the presented numbers.

$$(s_{k+1})^2 \geq (d_1 + \cdots + d_k)^2 \geq (d_1)^2 + \cdots + (d_k)^2 \ .$$

Based on this approximation we are convinced that a lemma analogous to Lemma 1 exists. However, there an ordering of the tasks $T = \{t_1, \ldots, t_n\}$ is required such that $\frac{(d_i)^2}{\alpha_i} \leq \frac{(d_j)^2}{\alpha_j}$ holds for $1 \leq i < j \leq n$. This and all the consequences have to be formally proved and their practical relevance has to be examined, too.

References

[1] Artiouchine, K., Baptiste, P.: Inter-distance constraint: An extension of the all-different constraint for scheduling equal length jobs. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 62–76. Springer, Heidelberg (2005)

[2] Baptiste, P., le Pape, C., Nuijten, W.: Constraint-Based Scheduling. International Series in Operations Research & Management Science, vol. 39. Kluwer Academic Publishers, Dordrecht (2001)

[3] Beldiceanu, N., Carlsson, M.: Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 377–391. Springer, Heidelberg (2001)

[4] Beldiceanu, N., Contjean, E.: Introducing global constraints in CHIP. Mathematical and Computer Modelling 12, 97–123 (1994)

[5] Caseau, Y., Laburthe, F.: Improved CLP scheduling with task intervals. In: van Hentenryck, P. (ed.) Proceedings of the Eleventh International Conference on Logic Programming, ICLP 1994, pp. 369–383. MIT Press, Cambridge (1994)

[6] Hoche, M., Müller, H., Schlenker, H., Wolf, A.: firstcs - A Pure Java Constraint Programming Engine. In: Hanus, M., Hofstedt, P., Wolf, A. (eds.) 2nd International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL 2003, September 29 (2003), `uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf`

[7] Régin, J.-C., Rueher, M.: A global constraint combining a sum constraint and difference constraints. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 384–395. Springer, Heidelberg (2000)

[8] Schulte, C., Stuckey, P.J.: When do bounds and domain propagation lead to the same search space. In: Sϕndergaard, H. (ed.) Third International Conference on Principles and Practice of Declarative Programming, Florence, Italy, September 2001, pp. 115–126. ACM Press, New York (2001)

[9] Van Hentenryck, P., le Provost, T.: Incremental search in constraint logic programming. New Generation Computing 9(3&4), 257–275 (1991)

[10] Vilím, P.: $O(n \log n)$ filtering algorithms for unary resource constraint. In: Régin, J.-C., Rueher, M. (eds.) CPAIOR 2004. LNCS, vol. 3011, pp. 335–347. Springer, Heidelberg (2004)

[11] Vilím, P.: Computing explanations for the unary resource constraint. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 396–409. Springer, Heidelberg (2005)

[12] Vilím, P., Barták, R., Čepek, O.: Unary Resource Constraint with Optional Activities. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 62–76. Springer, Heidelberg (2004)

[13] Wolf, A.: Pruning while sweeping over task intervals. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 739–753. Springer, Heidelberg (2003)

[14] Wolf, A.: Reduce-to-the-opt – A specialized search algorithm for contiguous task scheduling. In: Apt, K.R., Fages, F., Rossi, F., Szeredi, P., Váncza, J. (eds.) CSCLP 2003. LNCS, vol. 3010, pp. 223–232. Springer, Heidelberg (2004)

[15] Wolf, A.: Better propagation for non-preemptive single-resource constraint problems. In: Faltings, B.V., Petcu, A., Fages, F., Rossi, F. (eds.) CSCLP 2004. LNCS, vol. 3419, pp. 201–215. Springer, Heidelberg (2005)

Efficient Edge-Finding on Unary Resources with Optional Activities

Revised and Extended Version

Sebastian Kuhnert

Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
kuhnert@informatik.hu-berlin.de

Abstract. Unary resources play a central role in modelling scheduling problems. Edge-finding is one of the most popular techniques to deal with unary resources in constraint programming environments. Often it depends on external factors if an activity will be included in the final schedule, making the activity optional. Currently known edge-finding algorithms cannot take optional activities into account. This paper introduces an edge-finding algorithm that finds restrictions for enabled *and* optional activities. The performance of this new algorithm is studied for modified job-shop and random-placement problems.

Keywords: constraint-based scheduling, global constraints, optional tasks and activities, unary resources.

1 Unary Resources with Optional Activities

Many everyday scheduling problems deal with allocation of resources: Schools must assign rooms to lectures, factories must assign machines to tasks, train companies must assign tracks to trains. These problems are of high combinatorial complexity: For most of them there is no known polynomial time algorithm to find an optimal solution. Constraint programming offers a way to solve many instances of these problems in acceptable time [1].

Often an activity on a resource must be finished before a deadline and cannot be started before a release time. In this paper the following formalisation of scheduling problems is used: An **activity** (or task) i is described by its duration p_i and its earliest and latest starting and completion times (abbreviated as est_i, lst_i, ect_i, lct_i, respectively) as shown in Fig. 1. In constraint programming systems the start time is usually stored as a variable ranging from est_i to lst_i. The duration is constant in most applications; in this case the completion times can be derived as $\mathrm{ect}_i = \mathrm{est}_i + \mathrm{p}_i$ and $\mathrm{lct}_i = \mathrm{lst}_i + \mathrm{p}_i$. Otherwise let p_i denote the minimum duration for the purposes of this paper.[1]

[1] The values of p_i are used to restrict the domains of other variables. Using larger values than the minimum duration might cause unwarranted restrictions.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 38–53, 2009.

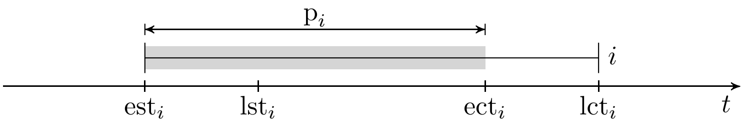


Fig. 1. Attributes of an activity i

Often sets Θ of activities are considered. The notions of production time and earliest starting time easily generalise to this case:

$$p_\Theta := \sum_{i \in \Theta} p_i \qquad\qquad est_\Theta := \min\{est_i \mid i \in \Theta\}$$

For $\Theta = \emptyset$ define $p_\emptyset := 0$ and $est_\emptyset := -\infty$.

A resource is called **unary resource**, if it can process only one activity at once and tasks cannot be interrupted. Now consider the following problem: Is there a valid schedule for a unary resource that includes all activities of a given set T, i. e. can the activities be assigned pairwise disjoint time periods satisfying the constraints given by the respective est_i, lst_i, p_i, ect_i and lst_i? See Fig. 2 for a sample instance. When the set T of activities grows, the number of possibilities to order them grows exponentially; it is long known that the unary resource scheduling problem is NP-complete [4, p. 236]. In constraint programming, the number of considered activity orderings is reduced in the following way: For each possible ordering of activities i and j (i before j or j before i), *restrictions* for the placement of other activities are derived before any other ordering decision is made. The process of searching for such restrictions is called *filtering*. Whenever a restriction is found, values can be removed from the domains of the variables describing the restricted activity. The goal of filtering is to reduce the number of ordering choices and to make the overall process of finding a solution faster. For this reason filtering algorithms should run fast and find as many restrictions as possible. On the other hand each restriction must be *justified*, i. e. there must not exist any solution with the removed values. In some cases filtering can find so many restrictions that the domain of a variable becomes empty, i. e. no possible schedule remains. This is called an *inconsistency* and causes backtracking in the search for possible orderings.

This work deals with **optional activities** on unary resources. These are tasks for which it is not yet known if they should be included in the final schedule:

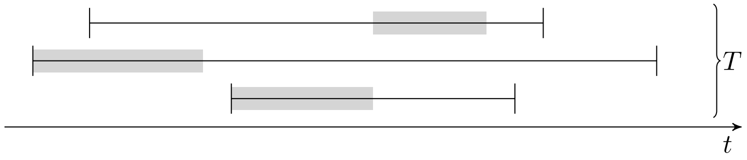


Fig. 2. A valid schedule for the unary resource with the tasks in T

E. g. when one specific task has to be performed once but several resources could do it. In this case optional activities are added to each resource, with the additional constraint that exactly one of them should be included in the final schedule.

Optional activities are modelled by adding another attribute: The variable status_i can take the values 1 for enabled and 0 for disabled. The domain $\{0, 1\}$ thus indicates an optional task. Additionally, let T_{enabled}, T_{optional} and T_{disabled} denote the partition of T into the enabled, optional and disabled tasks.

Optional activities entail additional difficulties for constraint filtering algorithms: As optional activities might become disabled later, they may not influence any other activity, because the resulting restrictions would not be justified. However, it is possible to detect if the inclusion of an optional activity causes an overload on an otherwise non-overloaded unary resource. In this case it is possible to disable this optional task. Doing this as often as possible while still maintaining fast filtering times is a desirable way to speed up the overall search process, because fewer ordering choices have to be enumerated later.

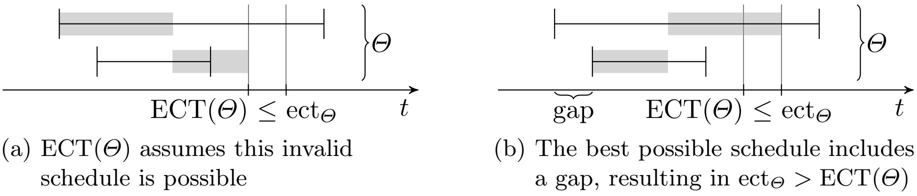
Several standard filtering algorithms for unary resources have been extended for optional activities by Vilím, Barták and Čepek [7]. To the best of the author's knowledge, no edge-finding algorithm has been proposed so far that considers optional activities. This paper aims to fill this gap.

2 Edge-Finding

There are two variants of edge-finding: One restricts the earliest starting times, the other restricts the latest completion times of the tasks on a unary resource. This paper only presents the former one, as the latter is symmetric to it.

To state the edge-finding rule and the algorithm, the following notational abbreviations are needed. Given a set Θ of n tasks, $\text{ECT}(\Theta)$ is a lower bound of the earliest completion time of all tasks in Θ:

$$\text{ECT}(\Theta) := \max_{\Theta' \subseteq \Theta} \{\text{est}_{\Theta'} + \text{p}_{\Theta'}\} \tag{1}$$


Θ
$\text{ECT}(\Theta) \leq \text{ect}_\Theta$
t
Θ
gap
$\text{ECT}(\Theta) \leq \text{ect}_\Theta$
t

(a) $\text{ECT}(\Theta)$ assumes this invalid schedule is possible

(b) The best possible schedule includes a gap, resulting in $\text{ect}_\Theta > \text{ECT}(\Theta)$

Fig. 3. $\text{ECT}(\Theta)$ can underestimate the real earliest completion time ect_Θ

The definition of $\mathrm{ECT}(\Theta)$ ignores gaps that may be necessary for a valid schedule, as illustrated in Fig. 3. This simplification results in weaker domain restrictions, but makes the computation of $\mathrm{ECT}(\Theta)$ feasible in the first place.[2]

Vilím has shown how the value $\mathrm{ECT}(\Theta)$ can be computed using a so-called Θ-tree [6], that has $\mathcal{O}(\log n)$ overhead for adding or removing tasks from Θ and offers constant time access to this value in return.

Later, extended this data structure was extended by Vilím, Barták and Čepek to Θ-Λ-trees. They retain the time complexity and include up to one task from an additional set Λ (disjoint from Θ), such that $\mathrm{ECT}(\Theta, \Lambda)$, the lower bound of the earliest completion time, becomes maximal [7]:

$$\mathrm{ECT}(\Theta, \Lambda) := \max_{j \in \Lambda} \{\mathrm{ECT}(\Theta \cup \{j\})\} \overset{(1)}{=} \max_{j \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{j\}} \{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'}\} \right\} \tag{2}$$

The purpose of this extension is to choose, for a given set Θ, one activity $j \in \Lambda$ such that $\mathrm{ECT}(\Theta \cup \{j\})$ becomes maximal. This is needed to efficiently find all applications of the edge-finding rule given below.

To handle optional activities, another extension is needed: The maximum (lower bound of the) earliest completion time for the tasks in Θ plus exactly one task from Ω plus up to one task from Λ (where Θ, Ω and Λ are pairwise disjoint):

$$\mathrm{ECT}(\Theta, \Omega, \Lambda) := \max_{o \in \Omega} \left\{ \max_{j \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{j\}} \left\{ \mathrm{est}_{\Theta' \cup \{o\}} + \mathrm{p}_{\Theta' \cup \{o\}} \right\} \right\} \right\} \tag{3}$$

As the algorithm presented below calculates the third argument to $\mathrm{ECT}(\Theta, \Omega, \Lambda)$ dynamically,[3] this value cannot be pre-computed using a tree structure. Instead, a loop over all tasks is needed, yielding $\mathcal{O}(n)$ time complexity. This loop iterates over all tasks $j \in T$, pre-sorted by their earliest starting time est_j, and keeps track of the earliest completion time of the tasks processed so far. To achieve this, a fact about partitions of task sets is used:

Definition 1 (Partition at an earliest starting time). *Given a set T of tasks, a pair of sets (L, R) is called* partition of T at the earliest starting time t_0*, iff (L, R) is a partition of T (i. e. $L \cup R = T$, $L \cap R = \emptyset$) that fulfils the following two conditions: $\forall l \in L : \mathrm{est}_l \leq t_0$ and $\forall r \in R : \mathrm{est}_r \geq t_0$.*

Proposition 1 (Calculation of ECT[4]). *Given a set T of tasks, and a partition (L, R) of T at an arbitrary earliest starting time, the following holds:*

$$\mathrm{ECT}(T) = \max \left\{ \mathrm{ECT}(R), \mathrm{ECT}(L) + \sum_{j \in R} \mathrm{p}_j \right\}$$

[2] If the exact earliest completion time ect_Θ of Θ could be calculated in polynomial time, the NP-complete problem "is there a valid schedule for this set of tasks" could be decided in polynomial time as well.

[3] The membership in Λ is not stored in memory, but decided by evaluating a condition. This also applies to one occurrence of $\mathrm{ECT}(\Theta, \Lambda)$.

[4] This is a variant of a proposition proven by Vilím, Barták and Čepek [7, p. 405] that was originally stated for left and right subtrees in a $\boldsymbol{\Theta}$-tree. The proof given there directly carries over to this proposition.

When iterating over the tasks $j \in T$, let L be the set of tasks considered so far (thus $\mathrm{ECT}(L)$ is known[5]) and $R := \{j\}$ the set containing only the current activity (for which $\mathrm{ECT}(R) = \mathrm{ect}_j$ is also known). Furthermore let $L' := L \cup \{j\}$ denote the set of activities considered after the current iteration. This way L grows from the empty set until it includes all activities in T. To compute not only $\mathrm{ECT}(\Theta)$ but $\mathrm{ECT}(\Theta, \Omega, \Lambda)$ on the basis of Proposition 1, the following values must be updated in the loop for each task j that is considered:

1. (The lower bound of) the earliest completion time of the Θ-activities only, i. e. $\mathrm{ECT}(L' \cap \Theta)$:

$$ect_{L'} := \begin{cases} \max\{ect_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \\ ect_L & \text{otherwise} \end{cases}$$

2. (The lower bound of) the earliest completion time when including up to one Λ-activity, i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Lambda)$:

$$ectl_{L'} := \begin{cases} \max\{ectl_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \\ \max\{ectl_L, \mathrm{ect}_j, ect_L + \mathrm{p}_j\} & \text{if } j \in \Lambda \\ ectl & \text{otherwise} \end{cases}$$

3. (The lower bound of) the earliest completion time when including exactly one Ω-activity (provided there is one), i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Omega, \emptyset)$:

$$ecto_{L'} := \begin{cases} \max\{ecto_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \wedge \Omega \cap L = \emptyset \\ ecto_L + \mathrm{p}_j & \text{if } j \in \Theta \wedge \Omega \cap L \neq \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j\} & \text{if } j \in \Omega \wedge \Omega \cap L = \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ecto_L\} & \text{if } j \in \Omega \wedge \Omega \cap L \neq \emptyset \\ ecto_L & \text{otherwise} \end{cases}$$

4. (The lower bound of) the earliest completion time when including up to one Λ- and exactly one Ω-activity (provided there is one), i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Omega, L' \cap \Lambda)$:

$$ectol_{L'} := \begin{cases} \max\{ectol_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \wedge \Omega \cap L = \emptyset \\ ectol_L + \mathrm{p}_j & \text{if } j \in \Theta \wedge \Omega \cap L \neq \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ectl_L + \mathrm{p}_j\} & \text{if } j \in \Omega \wedge \Omega \cap L = \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ectl_L + \mathrm{p}_j, ectol_L\} & \text{if } j \in \Omega \wedge \Omega \cap L \neq \emptyset \\ \max\{ectol_L, ecto_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Lambda \wedge \Omega \cap L = \emptyset \\ \max\{ectol_L, ecto_L + \mathrm{p}_j\} & \text{if } j \in \Lambda \wedge \Omega \cap L \neq \emptyset \\ ectol_L & \text{otherwise} \end{cases}$$

[5] For $L = \emptyset$ define $\mathrm{ECT}(\emptyset) := -\infty$.

After the last iteration (i. e. $L' = T$) $ectol_{L'} = \mathrm{ECT}(L' \cap \Theta, L' \cap \Lambda, L' \cap \Omega) = \mathrm{ECT}(\Theta, \Omega, \Lambda)$ contains the result of the computation.

Note that the indices of ect_L, $ectl_L$, $ectl_L$ and $ectol_L$ are only added for conceptual clarity. If the values are updated in the right order (i. e. $ectol_L$ first and ect_L last), only plain variables ect, $ectl$, $ecto$ and $ectol$ (and no arrays) are needed.

Finally, one more notational abbreviation is needed to state the edge-finding rule: The set of all tasks in a set T (usually all tasks or all enabled tasks of a resource), that end not later than a given task j:

$$\Theta(j,T) := \{k \in T \mid \mathrm{lct}_k \leq \mathrm{lct}_j\} \tag{4}$$

Notice the overloading of Θ; the return value of the function Θ corresponds to value of the set Θ at the time when j is processed in the main loop of the algorithm introduced later.

Rule 1 (Edge-Finding). *For all tasks $j \in T$ and $i \in T \setminus \Theta(j,T)$ holds: If*

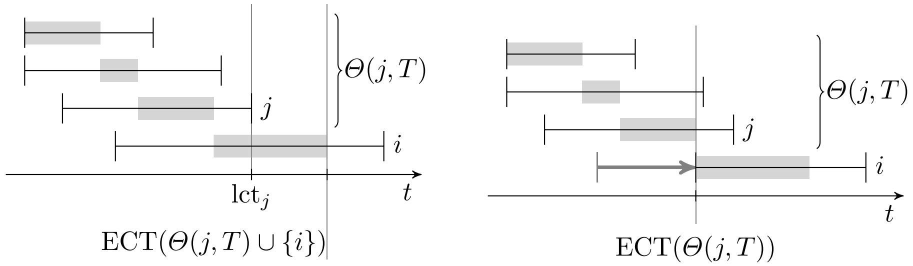
$$\mathrm{ECT}(\Theta(j,T) \cup \{i\}) > \mathrm{lct}_j \tag{5}$$

then i must be scheduled after all activities in $\Theta(j,T)$, i. e. the following restriction is justified:

$$\mathrm{est}_i \leftarrow \max\{\mathrm{est}_i, \mathrm{ECT}(\Theta(j,T))\} \tag{6}$$

This form of the edge-finding rule was introduced by Vilím, Barták and Čepek [7, p. 412ff] and proven equivalent to the more traditional form. The idea behind this rule is to detect if an activity i has to be the last one within the set $\Theta(j,T) \cup \{i\}$. This is ensured by the precondition (5) of the rule, which is illustrated in Fig. 4(a): As the latest completion time lct_j is by (4) also the latest completion time of $\Theta(j,T)$, the precondition states that $\Theta(j,T)$ must be finished before $\Theta(j,T) \cup \{i\}$ can be finished.

The resulting restriction (6), which is illustrated in Fig. 4(b), ensures that i is not started until all activities in $\Theta(j,T)$ can be finished.



(a) The rule is applicable for j and i: $\mathrm{ECT}(\Theta(j,T) \cup \{i\}) > \mathrm{lct}_j$

(b) After the application est_i is at least $\mathrm{ECT}(\Theta(j,T))$

Fig. 4. An application of the edge-finding rule

Besides the edge-finding rule, the presented edge-finding algorithm makes use of the following overload rule, that can be checked along the way without much overhead:

Rule 2 (Overload). *For all $j \in T$ holds: If* $\mathrm{ECT}(\Theta(j,T)) > \mathrm{lct}_j$ *then an overload has occurred, i. e. it is not possible to schedule all activities in T without conflict.*

This rule is illustrated in Fig. 5. The intuition for the overload rule is that the tasks in $\Theta(j,T)$ cannot possibly be finished before they have to.

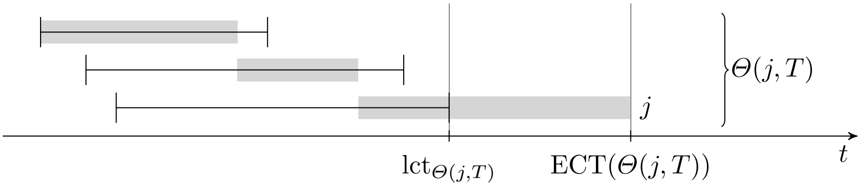


Fig. 5. An application of the overload rule

2.1 Edge-Finding Algorithm

Now to the central algorithm of this article: Listing 1 shows a program that finds all applications of the edge-finding rule to the set of enabled tasks and furthermore disables optional tasks whose inclusion would cause an overload (i. e. est_i could be raised to a value greater than lst_i for some task i).

Throughout the repeat-loop the set Θ is updated to reflect $\Theta(j, T_{\mathrm{enabled}})$ and Ω is updated to reflect $\Theta(j, T_{\mathrm{optional}})$ – exceptions are only allowed if multiple tasks have the same lct value. As j iterates over Q, lct_j decreases and j is removed from Θ or Ω and added to Λ.

Lines 15 to 25 handle enabled activities j and correspond closely to the algorithm by Vilím, Barták and Čepek [7, p. 416], with the only change being the handling of $\mathrm{status}_i = \mathrm{optional}$ in lines 20 to 24: If $\mathrm{ECT}(\Theta) > \mathrm{lst}_i$ (line 21) holds, the restriction $\mathrm{est}'_i \leftarrow \mathrm{ECT}(\Theta)$ (line 24) would cause an inconsistency as no possible start time for i remains. In this case the activity i is set to disabled (line 22), which fails for enabled activities and is the proper restriction for optional ones.

There are several more additions to handle optional activities: Firstly, the case $\mathrm{status}_j = \mathrm{optional}$ is handled. It follows the scheme of the enabled case, bearing in mind that j is optional: The overload-condition $\mathrm{ECT}(\Theta) > \mathrm{lct}_j$ on line 15 carries over to $\mathrm{ECT}(\Theta \cup \{j\}) > lct_j$ on line 39, where no immediate failure is generated but the optional activity j which would cause the overload is disabled.

If j is optional, the condition $\mathrm{ECT}(\Theta, \Lambda) > lct_j$ of the while-loop on line 18 can only result in the disabling of j as no optional activity may influence others. For this, no while-loop is required and a simple if-condition suffices. It must however take care to choose an appropriate $i \in \Lambda$ that leads to j being disabled. This is achieved by requiring $i \in T_{\mathrm{enabled}}$ and $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})$ in the condition on line 40.

Listing 1. Edge-finding algorithm

```
1   input: activities $T$ on a unary resource
2           // read-write access to $\mathrm{est}_i, \mathrm{lst}_i, \mathrm{lct}_i, \mathrm{status}_i$ is assumed for all $i \in T$
3   for $i \in T$ do // cache changes to $\mathrm{est}_i$: changing it directly would mess up $\Theta$-$\Lambda$-
        trees
4       $\mathrm{est}'_i \leftarrow \mathrm{est}_i$
5
6   $(\Theta, \Omega, \Lambda) \leftarrow (T_{\mathrm{enabled}}, T_{\mathrm{optional}}, \emptyset)$ // initialise tree(s)
7   $Q \leftarrow$ queue of all non-disabled $j \in T \setminus T_{\mathrm{disabled}}$ in descending order of $\mathrm{lct}_j$
8   $j \leftarrow Q.\mathit{first}$ // go to the first task in $Q$
9   repeat
10      if $\mathrm{status}_j \neq$ disabled then // move $j$ to $\Lambda$
11          $(\Theta, \Omega, \Lambda) \leftarrow (\Theta \setminus \{j\}, \Omega \setminus \{j\}, \Lambda \cup \{j\})$
12      $Q.\mathit{dequeue}$; $j \leftarrow Q.\mathit{first}$ // go to the next task in $Q$
13
14      if $\mathrm{status}_j =$ enabled then
15          if $\mathrm{ECT}(\Theta) > \mathrm{lct}_j$ then
16              fail // because overload rule (Rule 2) applies
17
18          while $\mathrm{ECT}(\Theta, \Lambda) > \mathrm{lct}_j$ do
19              $i \leftarrow$ the $\Lambda$-activity responsible for $\mathrm{ECT}(\Theta, \Lambda)$
20              if $\mathrm{ECT}(\Theta) > \mathrm{est}_i$ then // edge-finding rule is applicable
21                  if $\mathrm{ECT}(\Theta) > \mathrm{lst}_i$ then // inconsistency detected
22                      $\mathrm{status}_i \leftarrow$ disabled // fails if $i$ is enabled
23                  else // apply edge-finding rule
24                      $\mathrm{est}'_i \leftarrow \mathrm{ECT}(\Theta)$ // $\mathrm{ECT}(\Theta) > \mathrm{est}_i$ already ensured
25              $\Lambda \leftarrow \Lambda \setminus \{i\}$ // no better restriction for $i$ possible [7, p. 416]
26
27          while $\mathrm{ECT}(\Theta, \Omega) > \mathrm{lct}_j$ do // overload rule applies
28              $o \leftarrow$ the $\Omega$-activity responsible for $\mathrm{ECT}(\Theta, \Omega)$
29              $\mathrm{status}_o \leftarrow$ disabled
30              $\Omega \leftarrow \Omega \setminus \{o\}$
31
32          while $\Omega \neq \emptyset$ and $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o)) > \mathrm{lct}_j$ do // $\Lambda'(o)$ is defined in (7)
33              // edge-finding rule detects overload
34              $o \leftarrow$ the $\Omega$-activity responsible for $\mathrm{ECT}(\Theta, \Omega, \ldots)$
35                                      // already used in line 32 with that meaning
36              $\mathrm{status}_o \leftarrow$ disabled
37              $\Omega \leftarrow \Omega \setminus \{o\}$
38      else if $\mathrm{status}_j =$ optional then
39          if $\mathrm{ECT}(\Theta \cup \{j\}) > \mathrm{lct}_j$ // overload rule applicable ...
40          or $\mathrm{ECT}(\Theta \cup \{j\}, \{i \in T_{\mathrm{enabled}} \setminus \Theta \mid \mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})\}) > \mathrm{lct}_j$
41          then // ... or edge-finding rule detects overload
42              $\mathrm{status}_j \leftarrow$ disabled
43              $\Omega \leftarrow \Omega \setminus \{j\}$ // no more restrictions for $j$ possible
44  until the end of $Q$ is reached
45
46  for $i \in T$ do
47      $\mathrm{est}_i \leftarrow \mathrm{est}'_i$ // apply cached changes
```

Secondly, the case $\text{status}_j = \text{enabled}$ is extended with two more while-loops. They realise overload-detection and edge-finding for optional activities that are still contained in Ω. The set $\Lambda'(o)$, which is used in the condition of the second while-loop (line 32), is defined as follows (with T_{enabled} and Θ taken from the context where $\Lambda'(o)$ is evaluated):

$$\begin{aligned}\Lambda'(o) := \Big\{ i \in T_{\text{enabled}} \setminus \Theta \;\Big|\; & \text{lst}_i < \text{ECT}(\Theta \cup \{o\}) \\ & \wedge \nexists\, i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\text{lst}_{i_2} \geq \text{ECT}(\Theta \cup \{o\}) \\ & \qquad \wedge \text{ECT}(\Theta \cup \{i_2\}) > \text{ECT}(\Theta \cup \{i\}) \Big) \Big\} \end{aligned} \tag{7}$$

Like for the algorithm by Vilím, Barták and Čepek, the algorithm in Listing 1 must be repeated until it finds no further restrictions to make it idempotent. This issue is addressed by Listing 2 in Sect. 3 below.

Proposition 2 (Correctness). *The algorithm presented in Listing 1 is correct, i. e. all restrictions are justified.*

Proof. All restrictions are applications of the edge-finding and overload rules (Rules 1 and 2). It remains to be shown that the preconditions of the rules are met each time they are applied. This directly follows from the way the algorithm updates its data structures. As a simple example, consider the application of the overload rule in line 16. Remember that $\Theta = \Theta(j, T_{\text{enabled}})$. With the conditions in the preceding lines $j \in T_{\text{enabled}}$ and the precondition $\text{ECT}(\Theta(j, T_{\text{enabled}})) > \text{lct}_j$ is fulfilled.

For a more complicated case consider the disabling of the task o in line 36: This restriction is based on the edge-finding rule and that o is an optional activity. Remember that $\Theta = \Theta(j, T_{\text{enabled}})$ and $o \in \Omega = \Theta(j, T_{\text{optional}})$. Define $T' := T_{\text{enabled}} \cup \{o\}$. The activity i chosen from $\Lambda'(o)$ satisfies $i \in T_{\text{enabled}} \setminus \Theta$ by (7). This implies $i \in T' \setminus \Theta$. It holds that

$$\text{ECT}(\Theta, \Omega, \Lambda'(o)) = \text{ECT}(\Theta(j, T') \cup \{i\}) \,.$$

Together with the condition on the enclosing while loop this means that the precondition of the edge-finding rule is satisfied. From (7) also follows that $\text{lst}_i < \text{ECT}(\Theta \cup \{o\}) = \text{ECT}(\Theta(j, T'))$. This implies that the restriction of the edge-finding rule, if carried out, would lead to i having no possible start times left, i. e. an inconsistency would occur. As o is the only optional activity involved in this inconsistency, o cannot be included in a correct schedule and disabling it is correct.

The proofs for the other restrictions are similar. □

2.2 Complexity

Proposition 3 (Time Complexity). *The time complexity of the edge-finding algorithm presented in Listing 1 is $\mathcal{O}(n^2)$, where n is the number of activities on the unary resource.*

Proof. To see this, observe that each loop is executed at most n times: The two small copy-loops in lines 3–4 and 46–47 directly loop over all tasks, the main loop in lines 14–44 loops over the subset of non-disabled tasks and the three inner loops in lines 18–25, 27–30 and 32–37 each remove a task from Λ or Ω. As each task is added to these sets at most once, each loop is executed at most n times.

Sorting the tasks by lct (line 7) can be done in $\mathcal{O}(n \log n)$. All other lines (this includes all lines within the loops) bear at most $\mathcal{O}(n)$ complexity. Most are $\mathcal{O}(1)$, the more complicated ones are $\mathcal{O}(\log n)$ or $\mathcal{O}(n)$, as discussed for (1) to (3) at the beginning of Sect. 2.

Only the calculation of $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o))$ in line 32 needs special consideration, as o already references the task chosen from Ω. The definition of $\Lambda'(o)$ in (7) makes it possible to calculate it in the following way:

- When calculating $ectl_{L'}$ take *all* activities from $T_{\text{enabled}} \setminus \Theta$ in account, as it is not yet known which o will be chosen.
- For the cases with $j \in \Omega$ during the calculation of $ectol_{L'}$, consider $ectl_L + \mathrm{p}_j$ for calculating the maximum only if the activity i chosen for $ectl_L$ satisfies $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})$.
- For the cases with $j \in \Lambda$ during the calculation of $ectol_{L'}$, consider $ecto_L + \mathrm{p}_j$ for the maximum only if the activity o chosen for $ecto_L$ satisfies $\mathrm{lst}_j < \mathrm{ECT}(\Theta \cup \{o\})$.

To be able to access $\mathrm{ECT}(\Theta \cup \{o\})$ in constant time for all $o \in \Omega$, three arrays $ectAfter_o$, $pAfter_o$ and $ectBefore_o$ can be used, where after/before refers to the position of o in the list of tasks sorted by est_j and the three values consider only tasks from Θ. They can be computed based on Proposition 1 in linear time by looping over the tasks pre-sorted by est_j. It holds:

$$\mathrm{ECT}(\Theta \cup \{o\}) = \max\{ectAfter_o, \mathrm{ect}_o + pAfter_o, ectBefore_o + \mathrm{p}_o + pAfter_o\}$$

Thus the overall time complexity of the algorithm is $\mathcal{O}(n^2)$. □

Proposition 4 (Space Complexity). *The space complexity of the edge-finding algorithm presented in Listing 1 is $\mathcal{O}(n)$, where n is the number of activities on the unary resource.*

Proof. The algorithm uses only tree and list structures. Both have linear memory requirements. □

2.3 Optimality

For efficient restriction of the domains it is necessary that as many applications of the edge-finding rule as possible are found by the algorithm, while it is equally crucial that it offers fast runtimes.

Proposition 5 (Optimality). *The algorithm presented in Listing 1 finds all applications of the edge-finding rule to enabled activities and disables almost all optional activities that would cause an overload that is detectable by the edge-finding rule.*

Proof. First consider all applications of the edge-finding rule that involve only enabled tasks. Let therefore $j \in T_{\text{enabled}}$ and $i \in T_{\text{enabled}} \setminus \Theta(j, T_{\text{enabled}})$, such that $\text{ECT}(\Theta(j, T_{\text{enabled}}) \cup \{i\}) < \text{lct}_j$. As argued above, the set Θ at some point takes the value of $\Theta(j, T_{\text{enabled}})$ and i is contained in Λ. Then in line 20 i will be chosen (possibly after other i' that feature a larger $\text{ECT}(\Theta \cup \{i'\})$ have been chosen and removed from Λ) and est_i will be adjusted or an overload detected.

If an optional activity $o \in T_{\text{optional}}$ is involved, things are a bit more complicated. If the edge-finding rule is applicable for the set $T' := T_{\text{enabled}} \cup \{o\}$, the following cases can occur:

Case 1: $o \notin \Theta(j, T') \cup \{i\}$
The optional activity o is not involved and the edge-finding rule is equally applicable to the set T_{enabled}. The resulting restriction is found by the algorithm as argued above.

Case 2: $o = i$
This case is handled together with non-optional i in lines 18 to 25.

Case 3: $o = j$
In this case est_i may not be adjusted as the optional activity o may not influence the non-optional i. However, if est_i can be adjusted to a value greater than lst_i, the inclusion of o would cause an overload and o can be disabled. This is done in lines 38 to 43.

Case 4: $o \in \Theta(j, T') \setminus \{j\}$
Like in the previous case the only possible restriction is disabling o if it causes an overload. This case is handled in lines 32 to 37. For optimal results $\Lambda'(o)$ would have to be defined slightly differently, omitting the condition

$$\nexists\, i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\text{lst}_{i_2} \geq \text{ECT}(\Theta \cup \{o\}) \wedge \text{ECT}(\Theta \cup \{i_2\}) > \text{ECT}(\Theta \cup \{i\}) \Big) .$$

However, this would destroy the $\mathcal{O}(n^2)$ complexity of the algorithm. As this case never occurs for any of the problem instances measured in the next section, omitting these restrictions appears to be a good choice. □

3 Comparison with Other Edge-Finding Algorithms

The presented algorithm has been implemented for the Java constraint programming library `firstcs` [5] and compared to other algorithms. The following algorithms were considered:

New algorithm: The algorithm presented in Listing 1. It finds almost all possible applications of the edge-finding rule in $\mathcal{O}(n^2)$ time.

Simplified algorithm: Similar to the previous algorithm but without the loop in lines 32 to 37. This saves one of the two most expensive computations, though the asymptotic time complexity is still $\mathcal{O}(n^2)$. Fewer restrictions are found.

Active-only algorithm: The algorithm presented by Vilím, Barták and Čepek [7, p. 416]. It runs at $\mathcal{O}(n \log n)$ time complexity but takes no optional tasks into account.[6]

Iterative algorithm: Run the active-only algorithm once on T_{enabled} and then for each $o \in T_{\text{optional}}$ on $T_{\text{enabled}} \cup \{o\}$, adjusting or disabling o only. This algorithm finds all restrictions to optional activities at the cost of $\mathcal{O}(n^2 \log n)$ runtime.

To make these (and the other filtering algorithms, taken from [7]) idempotent, the fix-point technique of Listing 2 was used.

Listing 2. Fix-point iteration

```
repeat
    repeat
        repeat
            filter based on detectable precedences
        until no more restrictions are found
        filter with not-first/not-last rule
    until no more restrictions are found
    filter with the edge-finding algorithm to be benchmarked
until no more restrictions are found
```

For some benchmarks mentioned later, additional overload checking was performed by adding the following lines at the beginning of the innermost repeat-loop:

```
            if overload detected then
                fail
```

For now, this additional overload checking is skipped as all the studied edge-finding algorithms include applications of the overload rule. This way the comparison between the algorithms is more meaningful, as not only the differences in the way the edge-finding rule is applied, but also those in the way the overload rule is applied contribute to the results.

The search was performed using a modified resource-labelling technique. If optional tasks are allowed, the value of enabled_i has to be decided during the labelling as well. Usually it is decided at each level in the search tree in which order the corresponding pair of tasks $i, j \in T$ is to be executed. These two decisions were combined, so that the following alternatives are considered at each level: (a) both enabled with i before j, (b) both enabled with i after j, (c) only i enabled, (d) only j enabled or (e) both disabled.

[6] Although this algorithm is presented in an article mentioning optional activities in its title, it does not derive any restrictions from optional tasks. The reason it is listed in that paper is probably that it shares the data structure used, namely Θ-Λ-trees, with the other algorithms introduced there.

3.1 The Job-Shop Scheduling Problem

For benchmarking, *-alt* variants [7, p. 423] of job-shop problems [3] were used: For each job, exactly one of the fifth and sixth task must be included for a solution. All instances mentioned in this paper have 10 jobs each consisting of 10 tasks. This results in 10 machines with 10 tasks each. For the *-alt* instances 2 tasks per machine are initially optional on average. The last three instances contain no optional tasks and are included for reference. All times are measured in milliseconds and include finding a solution for the known optimal make-span and proving that this make-span is indeed optimal.

Table 1 shows the results. The number of backtracking steps required is equal for the new, the simplified and the iterative algorithm. So the new algorithm looses none of the possible restrictions, even if the loop in lines 32 to 37 is left out. When comparing the runtimes, the overhead of this loop is clearly visible: The simplified variant saves time, as it has lower overhead.

The active-only algorithm needs more backtracking steps because it finds less restrictions. Often the lower complexity compensates for this when it comes to the runtimes and the active-only variant is slightly better than the simplified one. The largest relative deviations is the other way around, though: For orb02-alt the active-only algorithm is almost 30% slower than the simplified one, for la19-alt it even needs 90% more time. These are also the instances with the largest difference in backtracking steps. It obviously depends on the inner structure of the job-shop instances (or the problems in general), if the higher time complexity of the newly proposed algorithms is justified by the backtracking steps saved by the additionally found restrictions.

Table 1. Runtime in milliseconds and number of backtracks for the different algorithms and job-shop-scheduling instances

	new		**simplified**		**active-only**		**iterative**	
instance	**time**	**bt**	**time**	**bt**	**time**	**bt**	**time**	**bt**
abz5-alt	4843	5859	4484	5859	4515	6441	4964	5859
orb01-alt	55747	56344	53662	56344	49361	56964	57013	56344
orb02-alt	7800	7265	7394	7265	9590	10610	7609	7265
orb07-alt	81856	99471	79063	99471	78309	104201	79786	99471
orb10-alt	136	85	125	85	121	85	132	85
la16-alt	7269	8294	6886	8294	6593	8841	7241	8294
la17-alt	46	9	31	9	35	11	31	9
la18-alt	26780	26846	25147	26846	24877	29039	25897	26846
la19-alt	2566	2022	2406	2022	4609	4632	2574	2022
la20-alt	62	53	63	53	55	53	62	53
abz5	5863	5107	5863	5107	5587	5107	5570	5107
orb01	29612	21569	29706	21569	28198	21569	28336	21569
orb02	10144	7937	10187	7937	9644	7937	9687	7937

The runtime of the iterative algorithm is mostly better than the new, but worse than the simplified algorithm.[7] As the tested instances all have relatively few optional activities per machine[8] (which benefits the active-only and iterative algorithms), it is interesting how the new algorithm performs for problems with more activities and a higher proportion of optional ones.

3.2 The Random Placement Problem

One such problem is the random placement problem [2], which can be solved using alternative resource constraints which in turn can be implemented using single resources with optional activities [8]. The runtimes of the instances provided on `http://www.fi.muni.cz/~hanka/rpp/` that are solvable in less than ten minutes are shown in Fig. 6.[9]

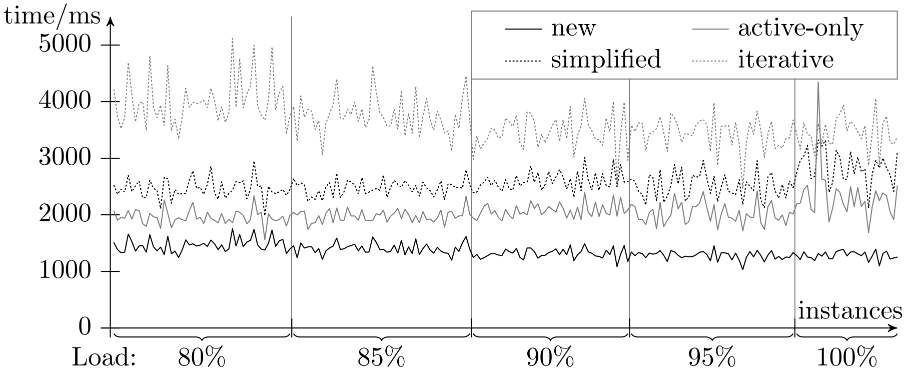


Fig. 6. Runtimes for instances of the Random-Placement-Problem, without additional overload checking

It is obvious that the new algorithm is the fastest in this case. Surprisingly the simplified algorithm is worse than the active-only one: Obviously it finds no or not enough additional restrictions. The power of the new algorithm thus derives from the loop omitted in the simplified one.[10] The iterative algorithm is the slowest, though it finds as many restrictions as the new one. The reason is the higher asymptotic time complexity, which seems to be prohibitive even

[7] An exception are the three last instances, which include no optional tasks.

[8] As mentioned above, the machines each have 10 tasks, 20% are optional on average.

[9] All algorithms need 0 backtracking steps, with the exception of the active-only algorithm for one instance with 100% load. This instance is responsible for the one peak going above even the iterative algorithm.

[10] This is confirmed by the number of choices needed during search: For the iterative and new algorithms it is always equal (the new one again looses no restrictions) and averagely 407, but the active-only and simplified ones both need more than 620 on average.

for relatively small resources with 50 to 100 tasks which appear in the tested random-placement instances.

So far only the fix-point algorithm without additional overload checking has been considered. This has its justification in the fact that all measured edge-finding algorithms perform overload-checking during their regular operation: By omitting the additional overload check their filtering strength contributes more to the results of the benchmark. In case of the job-shop benchmark, the additional overload checking furthermore results in slower runtimes for all algorithms. However, in case of the random placement benchmark, additional overload checking can lead to performance improvements for some edge-finding algorithms. This can be seen in Fig. 7: While the new and iterative algorithms loose 75 and 30 milliseconds, the active-only and simplified algorithms gain 990 and 1185 milliseconds averaged over the instances, overtaking the new algorithm. The additional overload checking thus can compensate for weaker performance of an edge-finding algorithm in this case.

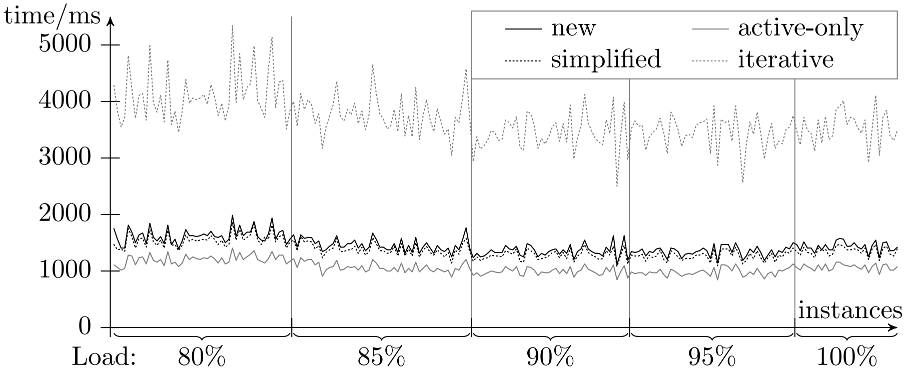


Fig. 7. Runtimes for instances of the Random-Placement-Problem, with additional overload checking

4 Summary

This paper introduces a new edge-finding algorithm for unary resources. It deals with optional activities correctly, finds all applications of the edge-finding rule to the enabled activities and disables optional activities if they cause an overload detected by the edge-finding rule. It is the first nontrivial edge-finding algorithm that derives restrictions for optional activities. It outperforms the naive, iterative implementation as soon as a larger fraction of optional activities is involved.

With $\mathcal{O}(n^2)$ asymptotic time complexity it is slower than edge-finding algorithms that cannot take optional activities into account. The additionally found restrictions offer a significant reduction of the search decisions needed to find and prove the solution of a problem. Especially for problems with many activities, of which a high proportion is optional, this can lead to faster runtimes. Performing additional overload checking can cancel this effect out, though.

Future work can study the influence of different labelling strategies when it comes to optional activities: Is it better to first establish an order of tasks and then decide which tasks should be enabled? Another possible direction for future work is studying which edge-finding algorithm yields best results depending on the problem structure, possibly implementing heuristics to decide which algorithm to use.

References

1. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling – Applying Constraint Programming to Scheduling Problems. Kluwer Academic Publishers, Dordrecht (2001)
2. Barták, R., Müller, T., Rudová, H.: A new approach to modeling and solving minimal perturbation problems. In: Apt, K.R., Fages, F., Rossi, F., Szeredi, P., Váncza, J. (eds.) CSCLP 2003. LNCS, vol. 3010, pp. 233–249. Springer, Heidelberg (2004)
3. Colombani, Y.: Constraint programming: an efficient and practical approach to solving the job-shop problem. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 149–163. Springer, Heidelberg (1996)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability – A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)
5. Hoche, M., Müller, H., Schlenker, H., Wolf, A.: firstcs– A Pure Java Constraint Programming Engine. In: Hanus, M., Hofstedt, P., Wolf, A. (eds.) 2nd International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL 2003, September 29 (2003), http://uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf
6. Vilím, P.: O(n log n) Filtering Algorithms for Unary Resource Constraint. In: Régin, J.-C., Rueher, M. (eds.) CPAIOR 2004. LNCS, vol. 3011, pp. 335–347. Springer, Heidelberg (2004), http://kti.mff.cuni.cz/~vilim/nlogn.pdf
7. Vilím, P., Barták, R., Čepek, O.: Extension of O(n log n) Filtering Algorithms for the Unary Resource Constraint to Optional Activities. In: Constraints, vol. 10.4, pp. 403–425 (2004), http://kti.mff.cuni.cz/~vilim/constraints2005.pdf
8. Wolf, A., Schlenker, H.: Realising the alternative resources constraint. In: Seipel, D., Hanus, M., Geske, U., Bartenstein, O. (eds.) INAP/WLP 2004. LNCS, vol. 3392, pp. 185–199. Springer, Heidelberg (2005)

Encoding of Planning Problems and Their Optimizations in Linear Logic

Lukáš Chrpa, Pavel Surynek, and Jiří Vyskočil

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague
{chrpa,surynek,vyskocil}@kti.mff.cuni.cz

Abstract. Girard's Linear Logic is a formalism which can be used to manage a lot of problems with consumable resources. Its expressiveness is quite good for an easily understandable encoding of many problems. We concentrated on expressing planning problems by linear logic in this paper. We observed a rich usage of a construct of consumable resources in planning problem formulations. This fact motivates us to provide a possible encoding of planning problems in linear logic. This paper shows how planning problems can be encoded in Linear Logic and how some optimizations of planning problems can be encoded. These optimizations can help planners to improve the efficiency of finding solutions (plans).

1 Introduction

Linear Logic is a formalism which can be used to formalize many problems with consumable resources [4]. There is a lot of such problems that handle with consumable and renewable resources in practice. Linear Logic gives us a good expressiveness that help us with formalization of these problems, because these problems can be usually encoded in formulae with linear size with respect to the length of the problems.

Previous research showed that planning problems can be encoded in many different formalisms, usually related to logic. It was showed in [16] that planning problems can be solved by reduction to SAT. This approach brings very good results (for example SATPLAN [17] won last two International Planning Competitions (IPC)[1] in optimal deterministic planning). However, there seems to be a problem with extending the SAT based planners to be able to solve planning problems with time and resources. One of the influential works regarding first order logic is a framework for programming reasoning agents called FLUX [26] which is based on Fluent Calculus. Another logic formalism which can be used in connection to planning problems is Temporal Logic (especially Simple Temporal Logic). It has been showed in [1] that Temporal Logic can be used for improvement of searching for plans. This approach is used in several planners (for example TALplanner [8] which gathered very good results in IPC 2002).

[1] http://ipc.icaps-conference.org

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 54–68, 2009.

Temporal Logic is good for representing relationships between actions, but not so good to represent whole planning problems.

Unlike other logics Linear Logic has a good expressiveness for formalization of whole planning problems including planning problems with resources etc. Linear Logic related research made during the last twenty years [12] brought many interesting results, however mainly in a theoretical area. One of the most important results was a connectivity of Linear Logic with Petri Nets [23] which gave us more sence of the good expressiveness of Linear Logic.

Instead of the encoding of Petri nets in Linear Logic, there is a possibility of encoding of planning problems in Linear Logic. Planning problems are an important branch of AI and they are very useful in many practical applications. Previous research showed that planning problems can be simply encoded in Linear Logic and a problem of plan generation can be reduced to a problem of finding a proof in linear logic. First ideas of solving planning problems via theorem proving in Linear Logic has been shown at [3,15,22]. These papers showed that M?LL fragment of Linear Logic is strong enough to formalize planning problems. Another interesting work in this area is [7] and it describes a possibility of a recursive application of partial plans. Analyzing planning problems encoded in Linear Logic via partial deduction approach is described in [19].

In the area of implementations of Linear Logic we should mention Linear Logic programming. Linear Logic Programming is derived from 'classical' logic programming (Prolog based) by including linear facts and linear operators. There are several Linear Logic programming languages, for example Lolli [14] or LLP [2]. However, all Linear Logic Programming languages are based on Horn's clauses which means that only one predicate can appear on the right side of implication. In fact, this forbids the obtaining of resources (resources can be only spent) which causes an inapplicability of Linear Logic programming languages in solving planning problems. Instead of Linear Logic programming languages there are several Linear Logic provers, for example llprover [24]. Linear Logic provers seem to be strong enough to solve planning problems, but the existing ones are very slow and practically almost unusable.

In the other hand, one of the most important practical results is an implementation of a planning solver called RAPS [20] which in comparison between the most successful planners in IPC 2002 showed very interesting results (especially in Depots domain). RAPS showed that the research in this area can be helpful and can provide an improvement for planners.

This paper extends the previous research in this area [5,22] by providing detailed description of encoding of planning problems into Linear Logic. In addition, we provide an encoding of many optimizations of planning problems, which helps a planner to find a solution more quickly. Furthermore, we designed two algorithms for actions assemblage that benefit from the Linear Logic encoding. Finally, we show how planning problems with resources can be encoded to Linear Logic and how Linear Logic can help in planning under uncertainty.

We provide a short introduction to Linear Logic and to planning problems in Section 2. Description of the pure encoding of planning problems in Linear

Logic and an extension of this encoding which works with negative predicates is provided in Section 3. Description of the encoding of optimizations such as encoding of static predicates, blocking actions, enforcing actions and assembling actions is provided in Section 4. Section 5 describes how planning with resources can be encoded in Linear Logic. Section 6 describes how planning under uncertainty can be encoded in Linear Logic. Section 7 concludes and presents a possible future research in this area.

2 Preliminaries

This section presents some basic information about Linear Logic and planning problems that helps the reader to get through this paper.

2.1 Linear Logic

Linear Logic was introduced by Girard in 1987 [10]. Linear Logic is often called 'logic of resources', because unlike the 'classical' logic, Linear Logic can handle with expendable resources. The main difference between 'classical' logic and Linear Logic results from an expression saying: "From A, A imply B we obtain B", in 'classical' logic A is still available after B is derived, but in Linear Logic A consumed after B is derived which means that A is no longer available (see **L**$\multimap$ rule in table 1).

In addition to the implication ($\multimap$), there are more operators in linear logic. However we mention only those that are relevant for our encoding. One of the operators is a multiplicative conjunction ($\otimes$) whose meaning is consuming (on the left side of the implication, see **L**$\otimes$ rule in table 1) or obtaining (on the right side of the implication, see **R**$\otimes$ rule in table 1) resources together. Another operator is an exponential (!), which converts a linear fact (expendable resource) to a 'classical' fact (not expendable resource). At last, there are an additive conjunction (&) and an additive disjunction ($\oplus$) whose meaning is close to modalities. Exactly when the additive conjunction is used on the right side of the implication, then only one alternative must be proved (see **R**& rule in table 1) , but when the additive disjunction is used on the right side of the implication, then all the alternatives must be proved (see **R**$\oplus$ rule in table 1). If the additives are on the left side of the implication, proving is quite similar, only the meaning of the additive conjunction and additive disjunction is swapped (see **L**& and **L**$\otimes$ rules in table 1).

Proving in Linear Logic is quite similar to proving in the 'classical' logic, which is based on Gentzen's style. Part of Linear Logic calculus needed to follow this paper is described in Table 1.The complete calculus of Linear Logic can be found in [10,11,21].

2.2 Planning Problems

This subsection brings us some basic introduction which is needed to understand basic notions frequently used in a connection to planning problems.

Table 1. Fragment of Linear Logic calculus

Id	$A \vdash A$	$\Gamma \vdash \top, \Delta$	**R**$\top$
L$\otimes$	$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, (A \otimes B) \vdash \Delta}$	$\frac{\Gamma_1 \vdash A, \Delta_1 \qquad \Gamma_2 \vdash B, \Delta_2}{\Gamma_1, \Gamma_2 \vdash (A \otimes B), \Delta_1 \Delta_2}$	**R**$\otimes$
L$\multimap$	$\frac{\Gamma_1 \vdash A, \Delta_1 \qquad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2, (A \multimap B) \vdash \Delta_1 \Delta_2}$		
L&	$\frac{\Delta, A \vdash \Gamma}{\Delta, (A \& B) \vdash \Gamma} \quad \frac{\Delta, B \vdash \Gamma}{\Delta, (A \& B) \vdash \Gamma}$	$\frac{\Delta, A \vdash \Gamma \qquad \Delta, B \vdash \Gamma}{\Delta, (A \oplus B) \vdash \Gamma}$	**L**$\oplus$
R&	$\frac{\Delta \vdash A, \Gamma \qquad \Delta \vdash B, \Gamma}{\Delta \vdash (A \& B), \Gamma}$	$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash (A \oplus B), \Delta} \quad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash (A \oplus B), \Delta}$	**R**$\oplus$
W !	$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta}$	$\frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta}$	**C !**
D !	$\frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}$	$\frac{\Gamma, A \vdash \Delta}{\Gamma, (\forall x) A \vdash \Delta}$	**Forall**

In this paper we consider action-based planning problems like Block World, Depots, Logistics etc. These planning problems are based on worlds containing objects (boxes, robots, etc.), locations (depots, platforms, etc.) etc. Relationships between objects and places (Box 1 is on Box2, Robot 1 is in Depot 2, etc.) are described by predicates. The worlds can be changed only by performing of actions. The next definitions define notions and notations well known in classical action-based planning problems.

Definition 1. *Assume that $L = \{p_1, \ldots, p_n\}$ is a finite set of predicates. Planning domain Σ over L is a 3-tuple (S, A, γ) where:*

- *$S \subseteq 2^L$ is a set of states. $s \in S$ is a state. If $p \in s$ then p is true in s and if $p \notin s$ then p is not true in s.*
- *A is a set of actions. Action $a \in A$ is a 4-tuple $(p^+(a), p^-(a), e^-(a), e^+(a))$ where $p^+(a) \subseteq L$ is a positive precondition of action a, $p^-(a) \subseteq L$ is a negative precondition of action a, $e^-(a) \subseteq L$ is a set of negative effects of action a and $e^+(a) \subseteq L$ is a set of positive effects of action a and $e^-(a) \cap e^+(a) = \emptyset$.*
- *$\gamma : S \times A \to S$ is a transition function. $\gamma(s, a) = (s - e^-(a)) \cup e^+(a)$ if $p^+(a) \subseteq s$ and $p^-(a) \cap s = \emptyset$.*

Remark 1. When a planning domain contains only actions without negative precondition we denote such an action by 3-tuple $(p(a), e^-(a), e^+(a))$ where $p(a)$ is a positive precondition.

Definition 2. Planning problem P *is a 3-tuple (Σ, s_0, g) such that:*

- *$\Sigma = (S, A, \gamma)$ is a planning domain over L.*
- *$s_0 \in S$ is an initial state.*
- *$g \subseteq L$ is a set of goal predicates.*

Definition 3. Plan π *is an ordered sequence of actions* $< a_1, \ldots, a_k >$ *such that, plan* π solves *planning problem* P *if and only if there exists an appropriate sequence of states* $< s_0, \ldots, s_k >$ *such that* $s_i = \gamma(s_{i-1}, a_i)$ *and* $g \subseteq s_k$. *Plan* π *is* optimal *if and only if for each* π' $\mid \pi \mid \leq \mid \pi' \mid$ *is valid (an optimal plan is the shortest plan solving a particular planning problem).*

To get deeper insight about planning problems, see [9].

3 Planning in Linear Logic

In this section, we show that Linear Logic is a good formalism for encoding planning problems. Main idea of the encoding is based on the fact that predicates in planning problems can be represented as linear facts that can be consumed or obtained depending on performed actions.

3.1 Basic Encoding of Planning Problems

Idea of a reduction of the problem of plan generation to finding a proof in Linear Logic was previously studied at [5,15,22]. At first we focus on such planning problems whose actions do not contain negative preconditions. As it was mentioned above, the predicates in planning problems can be encoded as the linear facts. Let $s = \{p_1, p_2, \ldots, p_n\}$ be a state, its encoding in Linear Logic is following:

$$(p_1 \otimes p_2 \otimes \ldots \otimes p_n)$$

Let $a = \{p(a), e^-(a), e^+(a)\}$ be an action, its encoding in Linear Logic is following:

$$\forall p_i \in p(a) \setminus e^-(a), 1 \leq i \leq l; \forall r_j \in e^-(a), 1 \leq j \leq m; \forall s_k \in e^+(a), 1 \leq k \leq n$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes r_1 \otimes r_2 \otimes \ldots \otimes r_m) \multimap (p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes s_1 \otimes s_2 \otimes \ldots \otimes s_n)$$

Performing of action a can be reduced to the proof in Linear Logic in following way ($\mathit{\Gamma}$ and $\mathit{\Delta}$ represent multiplicative conjunctions of literals and the vertical dots represent the previous part of the proof):

$$\dfrac{\dfrac{\vdots}{\mathit{\Gamma}, p_1, \ldots p_l, s_1, \ldots, s_n \vdash \mathit{\Delta}} \qquad \dfrac{}{p_1, \ldots p_l, r_1, \ldots, r_m \vdash p_1, \ldots p_n, r_1, \ldots, r_m}(Id)}{\mathit{\Gamma}, p_1, \ldots p_l, r_1, \ldots, r_m, ((p_1 \otimes \ldots \otimes p_l \otimes r_1 \otimes \ldots \otimes r_m) \multimap (p_1 \otimes \ldots \otimes p_l \otimes s_1 \otimes \ldots \otimes s_n)) \vdash \mathit{\Delta}} (L \multimap\ )$$

In most of planning problems it is not known how many times the actions are performed. This is the reason why the exponential ! is used for each rule representing the action. The proof must be modified in dependance of how many times the action is performed. There can be three cases:

- action a is not performed — then use $W!$ rule in a following way:

$$\frac{\mathit{\Gamma} \vdash \mathit{\Delta}}{\mathit{\Gamma}, !a \vdash \mathit{\Delta}} \, (W!)$$

- action a is performed just once — then use $D!$ rule in a following way:

$$\frac{\Gamma, a \vdash \Delta}{\Gamma, !a \vdash \Delta}\;(D!)$$

- action a is performed more than once — then use $D!$ and $C!$ rule in a following way:

$$\frac{\dfrac{\Gamma, !a, a \vdash \Delta}{\Gamma, !a, !a \vdash \Delta}\;(D!)}{\Gamma, !a \vdash \Delta}\;(C!)$$

The last thing which has to be explained is why a constant $\top$ must be used. The reason is that a goal state is reached when some state contains all the goal predicates. The state can certainly contain more predicates. The importance of the constant $\top$ can be seen in the following:

$$\frac{g_1, \ldots, g_n \vdash g_1 \otimes \ldots \otimes g_n \;(Id) \qquad s_1, \ldots, s_m \vdash \top \;(R\top)}{g_1, \ldots, g_n, s_1, \ldots, s_m \vdash g_1 \otimes \ldots \otimes g_n \otimes \top}\;(R\otimes)$$

Now it is clear that whole planning problem can be reduced to theorem proving in Linear Logic. Let $s_0 = \{p_{0_1}, p_{0_2}, \ldots, p_{0_m}\}$ be an initial state, $g = \{g_1, g_2, \ldots, g_q\})$ be a goal state and $a_1, a_2, \ldots, a_n$ be actions encoded as above. The whole planning problem can be encoded in a following way:

$$\begin{gathered}
p_{0_1}, p_{0_2}, \ldots, p_{0_m}, \\
!(p_1^1 \otimes p_2^1 \otimes \ldots \otimes p_{l_1}^1 \otimes r_1^1 \otimes r_2^1 \otimes \ldots \otimes r_{m_1}^1) \multimap (p_1^1 \otimes p_2^1 \otimes \ldots \otimes p_{l_1}^1 \otimes s_1^1 \otimes s_2^1 \otimes \ldots \otimes s_{n_1}^1), \\
!(p_1^2 \otimes p_2^2 \otimes \ldots \otimes p_{l_2}^2 \otimes r_1^2 \otimes r_2^2 \otimes \ldots \otimes r_{m_2}^2) \multimap (p_1^2 \otimes p_2^2 \otimes \ldots \otimes p_{l_2}^2 \otimes s_1^2 \otimes s_2^2 \otimes \ldots \otimes s_{n_2}^2), \\
\vdots \\
!(p_1^n \otimes p_2^n \otimes \ldots \otimes p_{l_n}^n \otimes r_1^n \otimes r_2^n \otimes \ldots \otimes r_{m_n}^n) \multimap (p_1^n \otimes p_2^n \otimes \ldots \otimes p_{l_n}^n \otimes s_1^n \otimes s_2^n \otimes \ldots \otimes s_{n_n}^n) \\
\vdash g_1 \otimes g_2 \otimes \ldots \otimes g_q \otimes \top
\end{gathered}$$

The plan exists if and only if the above expression is provable in Linear Logic. Obtaining of a plan from the proof can be done by checking of the $(L \multimap)$ rules from the bottom (the expression) to the top (axioms) of the proof.

3.2 Encoding of Negative Predicates

Previous subsection showed how planning problems can be encoded in Linear Logic. However, this encoding works with the positive precondition only. In planning problems there are usually used negative preconditions which means that an action can be performed if some predicate does not belong to the current state. However in Linear Logic the negative preconditions cannot be encoded directly. Fortunately, there are some possible approaches for bypassing of this problem.

The first approach can be used in propositional Linear Logic. The basic encoding of planning problems must be extended with linear facts representing negative predicates (each predicate p will obtain a twin $\overline{p}$ representing predicate $p \notin s$). It is clear that either p or $\overline{p}$ is contained in the each part of the proof. The encoding of state s, where predicates $p_1, \ldots, p_m \in s$ and predicates $p_{m+1}, \ldots, p_n \notin s$:

$$p_1 \otimes \ldots \otimes p_m \otimes \overline{p_{m+1}} \otimes \ldots \otimes \overline{p_n}$$

Each action $a = \{p^+(a), p^-(a), e^-(a), e^+(a)\}$ from a given planning domain can be transformed to action $a' = \{p'(a'), e'^-(a'), e'^+(a')\}$, where $p'(a') = p^+(a) \cup \{\overline{p}\,|p \in p^-(a)\}$, $e'^-(a') = e^-(a) \cup \{\overline{p}\,|p \in e^+(a)\}$ and $e'^+(a') = e^+(a) \cup \{\overline{p}\,|p \in e^-(a)\}$. The encoding of the action a' is following:

$$\forall p_i \in p'(a') \setminus e'^-(a'), 1 \leq i \leq l; \forall \overline{p_{i'}} \in p'(a') \setminus e'^-(a'), 1 \leq i' \leq l'$$
$$\forall r_j \in e'^-(a'), 1 \leq j \leq m; \forall s_k \in e'^+(a'), 1 \leq k \leq n$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes \overline{p_1} \otimes \overline{p_2} \otimes \ldots \otimes \overline{p_{l'}} \otimes r_1 \otimes r_2 \otimes \ldots \otimes r_m \otimes \overline{s_1} \otimes \overline{s_2} \otimes \ldots \otimes \overline{s_n}) \multimap$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes \overline{p_1} \otimes \overline{p_2} \otimes \ldots \otimes \overline{p_{l'}} \otimes s_1 \otimes s_2 \otimes \ldots \otimes s_n \otimes \overline{r_1} \otimes \overline{r_2} \otimes \ldots \otimes \overline{r_m})$$

Second approach can be used in predicate Linear Logic. Each linear fact representing predicate $p(x_1, \ldots, x_n)$ can be extended by one argument representing if predicate $p(x_1, \ldots, x_n)$ belongs to the state s or not ($p(x_1, \ldots, x_n) \in s$ can be represented as $p(x_1, \ldots, x_n, 1)$ and $p(x_1, \ldots, x_n) \notin s$ can be represented as $p(x_1, \ldots, x_n, 0)$). Encoding of actions can be done in a similar way like in the first approach. The advantage of this approach is in the fact that the representation of predicates can be generalized to such a case that more than one (same) predicate is available. It may be helpful in encoding of some other problems (for example Petri Nets).

3.3 Example

In this example we will use a predicate extension of Linear Logic. Imagine a version of "Block World", where we have slots and boxes, and every slot may contain at most one box. We have also a crane, which may carry at most one box.

Objects: 3 slots (1,2,3), 2 boxes (a, b), crane

Initial state: $in(a,1) \otimes in(b,2) \otimes free(3) \otimes empty$

Actions:

$$\begin{aligned} PICKUP(Box, Slot) = \{ \quad p &= \{empty, in(Box, Slot)\}, \\ e^- &= \{empty, in(Box, Slot)\}, \\ e^+ &= \{holding(Box), free(Slot)\}\} \end{aligned}$$

$$\begin{aligned} PUTDOWN(Box, Slot) = \{ \quad p &= \{holding(Box), free(Slot)\}, \\ e^- &= \{holding(Box), free(Slot)\}, \\ e^+ &= \{empty, in(Box, Slot)\}\} \end{aligned}$$

Goal: Box a in slot 2, Box b in slot 1.

The encoding of the action $PICKUP(Box, Slot)$ and $PUTDOWN(Box, Slot)$:

$PICKUP(Box, Slot) : empty \otimes in(Box, Slot) \multimap holding(Box) \otimes free(Slot)$

$PUTDOWN(Box, Slot) : holding(Box) \otimes free(Slot) \multimap empty \otimes in(Box, Slot)$

The whole problem can be encoded in Linear Logic in the following way:

$$in(a,1), in(b,2), free(3), empty, !(empty \otimes in(Box, Slot) \multimap holding(Box) \otimes free(Slot)), !(holding(Box) \otimes free(Slot) \multimap empty \otimes in(Box, Slot)) \vdash in(b,1) \otimes in(a,2) \otimes \top$$

3.4 Additional Remarks

The basic encoding is accurate if the following conditions are satisfied:

- $e^-(a) \subseteq p(a)$, $\forall a \in A$
- $p(a) \subseteq s$ and $s \cap e^+(a) = \emptyset$, $\forall a \in A$ and s is a state

Even though there are not many domains violating these conditions, the violation may cause that the basic encoding is inaccurate. If the first condition is violated, it may happen that some actions normally performable will not be performable via Linear Logic, because all predicates from negative effects are placed on the left hand side of the implication which means that all predicates must be presented before the performance of the action (regarding the definition we can perform every action if all predicates from its precondition are presented in a certain state). If the second condition is violated, it may happen that after the performance of the action violating this condition some predicates will be presented more times than once. To avoid these troubles we have to convert the basic encoding into the encoding supporting negative predicates and we must put $(p \oplus \overline{p})$ into left side of implications for every predicate p which may cause the breaking of the conditions. It is clear that either p or $\overline{p}$ is presented in every state. $(p \oplus \overline{p})$ on the left side of the implication ensures that either p or $\overline{p}$ is removed from the particular state (depending on which one is available) and then there is added p (if p is in positive effects) or $\overline{p}$ (if p is in negative effects).

4 Encoding Optimizations of Planning Problems in Linear Logic

In the previous section, we showed the pure encoding of planning problems in Linear Logic. To improve efficiency of the searching for a plan, it is needed to encode some optimizations described in the next subsections.

4.1 Handling with Static Predicates

Static predicates are often used in planning problems. Static predicates can be easily detected, because each static predicate in a planning problem is such predicate that belongs to an initial state and does not belong to any effect of any action (static predicates appear only in preconditions). It is possible to encode the static predicates like 'classical' facts using the exponential !. Assume action $a = \{p(a) = \{p_1, p_2\}, e^-(a) = \{p_1\}, e^+(a) = \{p_3\}$ where p_2 is a static predicate. Action a can be encoded in a following way:

$$(p_1 \otimes ! p_2) \multimap p_3$$

The encoding of static predicates is described in propositional Linear Logic for better understanding. This encoding has the purpose in predicate Linear Logic (in propositional Linear Logic static predicates can be omitted).

4.2 Blocking of Actions

To increase efficiency of solving planning problems it is quite necessary to use some technique which helps a solver to avoid unnecessary backtracking. Typically, the way how a lot of unnecessary backtracking can be avoided is blocking of actions that are not leading to a solution (for example inverse actions).

The idea how to encode the blocking of actions rests in an addition of new predicate $can(a, x)$, where a is an action and $x \in \{0, 1\}$ is representing a status of action a. If $x = 0$ then action a is blocked and if $x = 1$ then action a is unblocked. Now it is clear that the encoding of action a can be done in the following way: (Assume $a = \{p(a) = \{p_1\}, e^-(a) = \{p_1\}, e^+(a) = \{p_3\})$

$$\begin{gathered}(can(a, 1) \otimes p_1) \multimap (can(a, 1) \otimes p_3), \text{or} \\ (can(a, 1) \otimes p_1) \multimap (can(a, 0) \otimes p_3)\end{gathered}$$

The first expression means that a does not block itself and the second expression means that a blocks itself.

Assume that some action $b = \{p(b) = \{q_1\}, e^-(b) = \{q_1\}, e^+(b) = \{q_2\}$ can block action a. Encoding of action b is following ($can(b, ?)$ means $can(b, 1)$ or $can(b, 0)$ like in the previous encoding of action a):

$$\forall X : (can(b, 1) \otimes can(a, X) \otimes q_1) \multimap (can(b, ?) \otimes can(a, 0) \otimes q_2)$$

The predicate $can(a, X)$ from the expression above represents the fact that we do not known if a is blocked or not. If a is already blocked then X unifies with 0 and anything remains unchanged ($can(a, 0)$ still holds). If a is not blocked then X unifies with 1 which means that a become blocked, because $can(a, 1)$ is no longer true and $can(a, 0)$ become true.

Now assume that action $c = \{p(c) = \{r_1\}, e^-(c) = \{r_1\}, e^+(c) = \{r_2\}$ can unblock action a. Encoding of action c can be done in a similar way like before. The encoding of action c is following ($can(c, ?)$ means $can(c, 1)$ or $can(c, 0)$ like in the previous encoding of action a):

$$\forall X : (can(c, 1) \otimes can(a, X) \otimes r_1) \multimap (can(c, ?) \otimes can(a, 1) \otimes r_2)$$

The explanation how this encoding works is similar like the explanation in the previous paragraph.

4.3 Enforcing of Actions

Another optimization which can help a solver to find a solution faster is enforcing of actions. Typically, when some action is performed it is necessary to perform some other action. It is possible to enforce some action by blocking of other actions, but it may decrease the efficiency, because each single action must be blocked in the way described in the previous subsection which means that formulae rapidly increase their length.

The idea how enforcing of actions can be encoded rests also in an addition of new predicate $can(a)$, where a is an only action which can be performed. Let $can(1)$ represent the fact that all actions can be performed. The encoding of action $a = \{p(a) = \{p_1\}, e^-(a) = \{p_1\}, e^+(a) = \{p_3\}$ which does not enforce any other action is following:

$$((can(a) \oplus can(1)) \otimes p_1) \multimap (can(1) \otimes p_3)$$

The expression $can(a) \oplus can(1)$ means that action a can be performed if and only if a is enforced by other action ($can(a)$ is true) or all actions are allowed ($can(1)$ is true).

Now assume that action $b = \{p(b) = \{q_1\}, e^-(b) = \{q_1\}, e^+(b) = \{q_2\}$ enforces the action a. The encoding of action b is following:

$$((can(b) \oplus can(1)) \otimes q_1) \multimap (can(a) \otimes q_2)$$

It is clear that in this encoding there is one or all actions allowed in a certain step. This idea can be easily extended by adding some new symbols representing groups of allowed actions.

4.4 Assembling of Actions into a Single Action

Another kind of optimization in planning problems is assembling of actions into a single action, usually called macro-action (for deeper insight see [18]). This approach is based on a fact that some sequences of actions are used several times. Let $a_1, \ldots, a_n$ be a sequence of actions encoded in Linear Logic in the way described before. For further usage we use shortened notation of the encoding:

$$\bigotimes_{\forall l \in \Gamma_i} l \multimap \bigotimes_{\forall r \in \Delta_i} r \quad \forall i \in \{1, \ldots, n\}$$

Assume that action a is created by an assembling of a sequence of actions $a_1, \ldots, a_n$. Because a is also an action, the encoding is following:

$$\bigotimes_{\forall l \in \Gamma} l \multimap \bigotimes_{\forall r \in \Delta} r$$

The following algorithm shows how action a can be obtained from the sequence of actions $a_1, \ldots, a_n$.

Algorithm 1:
`INPUT:` $\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots, \Delta_n$ (see the previous encoding of actions $a_1, \ldots, a_n$)
`OUTPUT:` Γ, Δ (see the previous encoding of action a)

```
$\Gamma := \Delta := \emptyset$
for $i = 1$ to $n$ do
    $\Lambda := \Delta \cap \Gamma_i$
    $\Delta := (\Delta \setminus \Lambda) \cup \Delta_i$
    $\Gamma := \Gamma \cup (\Gamma_i \setminus \Lambda)$
endfor
```

Proposition 1. *Algorithm 1 is correct.*

Proof. The correctness of algorithm 1 can be proved inductively in a following way:

For $n = 1$, it is easy to see that algorithm 1 works correctly when only action a_1 is in the input. The for cycle on lines 2-6 runs just once. On line 3 it is easy to see that $\Lambda = \emptyset$, because $\Delta = \emptyset$. Now it can be seen that $\Delta = \Delta_1$ (line 4) and $\Gamma = \Gamma_1$ (line 5), because $\Lambda = \emptyset$ and Γ and Δ before line 4 (or 5) are also empty.

Assume that algorithm 1 works correctly for k actions. From this assumption imply existence of an action a that is created by algorithm 1 from some sequence of actions $a_1, \ldots, a_k$ after k steps of the for cycle in lines 2-6. Let s be a state and s' be a state which is obtained from s by applying action a (without loss of generality assume that a can be applied on s). It is true that $s' = (s \setminus \Gamma) \cup \Delta)$. Let a_{k+1} be an action which is applied on state s' (without loss of generality assume that a_{k+1} can be applied on s'). It is true that $s'' = (s' \setminus \Gamma_{k+1}) \cup \Delta_{k+1} = ((s \setminus \Gamma) \cup \Delta) \setminus \Gamma_{k+1}) \cup \Delta_{k+1}$. After $k+1$-th step of the for cycle (lines 2-6) action a' is created (from sequence $a_1, \ldots, a_k, a_k + 1$). When a' is applied on state s, $s''' = (s \setminus \Gamma') \cup \Delta')$. From lines 3-5 it can be seen that $\Gamma' = \Gamma \cup (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$ and $\Delta' = (\Delta \setminus (\Delta \cap \Gamma_{k+1})) \cup \Delta_{k+1}$. Now $s''' = (s \setminus (\Gamma \cup (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1})))) \cup ((\Delta \setminus (\Delta \cap \Gamma_{k+1})) \cup \Delta_{k+1})$. To finish the proof we neede to prove that $s'' = s'''$. $p \in s''$ iff $p \in \Delta_{k+1}$ or $p \in \Delta \wedge p \notin \Gamma_{k+1}$ or $p \in s \wedge p \notin \Gamma \wedge (p \notin \Gamma_{k+1} \vee p \in \Delta)$. $p \in s'''$ iff $p \in \Delta_{k+1}$ or $p \in \Delta \wedge p \notin (\Delta \cap \Gamma_{k+1})$ or $p \in s \wedge p \notin \Gamma \wedge p \notin (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$. It is easy to see that $p \in \Delta \wedge p \notin (\Delta \cap \Gamma_{k+1})$ is satisfied iff $p \in \Delta \wedge p \notin \Gamma_{k+1}$ is satisfied. $p \notin (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$ is satisfied iff $p \notin \Gamma_{k+1}$ or $p \in \Delta$ is satisfied. Now is clear that $s'' = s'''$. □

Algorithm 1 works with planning problems encoded in propositional Linear Logic. The extension of the algorithm to predicate Linear Logic can be simply done by adding of constraints symbolizing which actions' arguments must be equal. This extension affect only a computation of intersection $(\Delta_i \cap \Gamma_j)$.

The following algorithm for assembling of actions into a single action is based on a paradigm divide and conquer which can support a parallel implementation.

Algorithm 2:
`INPUT:` $\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots, \Delta_n$ (see the previous encoding of actions $a_1, \ldots, a_n$)
`OUTPUT:` Γ, Δ (see the previous encoding of action a)

```
Function Assemble($\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots \Delta_n$):$\Gamma, \Delta$
    if $n = 1$ then return($\Gamma_1, \Delta_1$) endif
    $\Gamma', \Delta'$:=Assemble($\Gamma_1, \ldots, \Gamma_{\lceil\frac{n}{2}\rceil}, \Delta_1, \ldots, \Delta_{\lceil\frac{n}{2}\rceil}$)
    $\Gamma'', \Delta''$:=Assemble($\Gamma_{\lceil\frac{n}{2}\rceil+1}, \ldots, \Gamma_n, \Delta_{\lceil\frac{n}{2}\rceil+1}, \ldots, \Delta_n$)
    $\Lambda := \Delta' \cap \Gamma''$
    $\Gamma := \Gamma' \cup (\Gamma'' \setminus \Lambda)$
    $\Delta := \Delta'' \cup (\Delta' \setminus \Lambda)$
    return($\Gamma, \Delta$)
endFunction
```

Proposition 2. *Algorithm 2 is correct.*

Proof. The correctness of algorithm 2 can be proved in a following way:

$n = 1$ — It is clear that the assemblage of one-element sequence of actions (a_1) is equal to action a_1 itself.

$n > 1$ — Let $a_1, \ldots a_n$ be a sequence of actions. In lines 2 and 3 the sequence splits into two sub-sequences ($a_1, \ldots, a_{\lceil\frac{n}{2}\rceil}$ and $a_{\lceil\frac{n}{2}\rceil+1}, \ldots, a_n$) and algorithm 2 is applied recursively on them. Because $\lceil\frac{n}{2}\rceil < n$ when $n > 1$, it is easy to see that the recursion will finitely terminate (it happens when $n = 1$). Now it is clear that a' and a'' are actions obtained by the assembling of sequences $a_1, \ldots, a_{\lceil\frac{n}{2}\rceil}$ and $a_{\lceil\frac{n}{2}\rceil+1}, \ldots, a_n$. These actions are assembled into a single action a at lines 4-6. The proof of the correctness of this assemblage is done in the proof of proposition 1.

□

Algorithm 2 can be extended to predicate Linear Logic in a similar way like algorithm 1.

5 Linear Logic in Planning with Resources

Planning problems with resources becomes commoner, because it has more practical applicability than classical planning. Linear Logic itself as mentioned before is often called 'logic of resources'. Even though propositional Linear Logic does not seem to be good for representing planning problems with resources. We are able to represent the number of units of resources by linear facts, but handling with them is difficult, making the encoding much more complex. The biggest problem is refilling of the resources to some predefined level, because in Linear Logic we usually cannot find out how many units of the resources are remaining. Instead of propositional Linear Logic we can use predicate Linear Logic where we are able to use function symbols like + or −. In addition, we can use comparative operators that can be represented in Linear Logic notation as binary predicates with exponential !. Here, all classical function symbols and comparative operators are written in classical infix form. Let

$a = \{p(a) = \{p_1\}, e^-(a) = \{p_1\}, e^+(a) = \{p_3\}$ be an action which in addition requires at least 3 units of resource r and after performance of a 3 units of r will be consumed. The encoding of a is following:

$$\forall X : (p_1 \otimes r(X) \otimes X \geq 3) \multimap (p_3 \otimes r(X-3))$$

Let $b = \{p(b) = \{q_1\}, e^-(b) = \{q_1\}, e^+(b) = \{q_2\}$ be an action which in addition refills resource r to 10 units. The encoding of b is following:

$$\forall X : (q_1 \otimes r(X)) \multimap (q_2 \otimes r(10))$$

Predicate r representing some resource must appear on the both sides of the implication, because in every state predicate r must be listed just once. The first rule can be applied if and only if p_1 is true and r contains at least 3 units. It is done in the left side of the implication where predicates p_1 and r are removed. In the right side of the implication we add predicates p_3 and r, where r is decreased by 3 units. The second rule can be applied if and only if q_1 is true (do not depend on r). It is done in the left side of the implication where predicates q_1 and r are removed. In the right side of the implication we add predicates q_2 and r, where r is set to 10 units.

6 Linear Logic in Planning under Uncertainty

The main difference between deterministic planning and planning under uncertainty is such that actions in planning under uncertainty can reach more states (usually we do not know which one). The main advantage of Linear Logic, which can be used in planning under uncertainty, are additive operators (&) and ($\oplus$). We have two options how to encode uncertain actions. In the following expressions we assume that after the performance of action a on state s we obtain one of $s_1, s_2, \ldots, s_n$ states in the certain steps (remember the encodings of states):

$$s \multimap (s_1 \& s_2 \& \ldots \& s_n)$$

$$s \multimap (s_1 \oplus s_2 \oplus \ldots \oplus s_n)$$

If we use additive conjunction (&) we want to find a plan which may succeed (with nonzero probability). Recall the rule $L\&$ which gives the choice which state will be set as following. If we use additive disjunction ($\oplus$) we want to find a plan which certainly succeeds. Recall the rule $L\oplus$ which tells that we have to proof that from all of the following states we can certainly reach the goal. Even though Linear Logic cannot handle probabilities well, we can use it for decision if there exists some plan which certainly succeed or if there is no chance to find some plan which may succeed.

7 Conclusions and Future Research

The previous research showed that Linear Logic is a good formalism for encoding many problems with expendable resources like planning problems, because in comparison to the other logics, Linear Logic seems to be strong enough to represent also planning problems with time and resources. This paper extends the previous research in this area by providing detailed description of encoding of planning problems into Linear Logic and by showing that many optimizations of planning problems, which helps a planner to find a solution more quickly, can be also easily encoded in Linear Logic. Main advantage of this approach also rests in the fact that an improvement of the Linear Logic solver leads to improved efficiency of the planner based on Linear Logic.

One of possible directions how to implement an efficient algorithm for solving planning problems encoded in Linear Logic is using the connection between Linear Logic and Petri Nets. It is not difficult to see that the encoding of the planning problems is similar to an encoding of Petri Nets. The implementation of an unfolding algorithm for reachability problem in Petri Nets for solving planning problems has been done by [13] and showed very good results. We will study the possibility of extending this algorithm in the way that the extended algorithm will support the encoding of planning problems in predicate Linear Logic. There is a possibility of extension of the presented encodings of planning problems in Linear Logic into Temporal Linear Logic (to learn more about Temporal Linear Logic, see [25]). It seems to be a very useful combination, because Temporal Logic can give us more possibilities for encoding of relationships between actions. Another possible way of our future research is using of the encodings of the optimizations for transformation of planning domains. Transformed planning domains can be used with existing planners and can reach better results, because the presented optimizations of planning problems can help the planners to prune the search space. We focused on getting knowledge from plan analysis which helps us to find out the optimizations. We studied it in [6] and results that were provided there are very interesting. At last, Linear Logic seems to be strong enough to encode planning problems with time. Basic idea of this extension comes from the fact that time can be also encoded into linear facts. The usage of predicate Linear Logic seems to be necessary in this case as it as in planning with resources.

Acknowledgements

We thank the reviewers for the comments. The research is supported by the Czech Science Foundation under the contracts no. 201/08/0509 and 201/05/H014 and by the Grant Agency of Charles University (GAUK) under the contract no. 326/2006/A-INF/MFF.

References

1. Bacchus, F., Kabanza, F.: Using temporal logics to express search control knowledge for planning. Artificial Intelligence 22, 5–27 (1998)
2. Banbara, M.: Design and Implementation of Linear Logic Programming Languages. Ph.D. Dissertation, The Graduate School of Science and Technology, Kobe University (2002)

3. Bibel, W., Cerro, L.F., Fronhofer, B., Herzig, A.: Plan Generation by Linear Proofs: On Semantics. In: Proceedings of GWAI, pp. 49–62 (1989)
4. Chrpa, L.: Linear Logic: Foundations, Applications and Implementations. In: Proceedings of workshop CICLOPS, pp. 110–124 (2006)
5. Chrpa, L.: Linear logic in planning. In: Proceedings of Doctoral Consorcium ICAPS, pp. 26–29 (2006)
6. Chrpa, L., Bartak, R.: Towards getting domain knowledge: Plans analysis through investigation of actions dependencies. In: Proceedings of FLAIRS, pp. 531–536
7. Cresswell, S., Smaill, A., Richardson, J.: Deductive Synthesis of Recursive Plans in Linear Logic. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS, vol. 1809, pp. 252–264. Springer, Heidelberg (2000)
8. Doherty, P., Kvanstrom, J.: TALplanner: A temporal logic based planner. AI Magazine 22(3), 95–102 (2001)
9. Ghallab, M., Nau, D., Traverso, P.: Automated planning, theory and practice. Morgan Kaufmann Publishers, San Francisco (2004)
10. Girard, J.-Y.: Linear logic. Theoretical computer science 50, 1–102 (1987)
11. Girard, J.-Y.: Linear Logic: Its Syntax and Semantics. Technical report, Cambridge University Press (1995)
12. Ehrhard, T., Girard, J.Y., Ruet, P., Scott, P.: Linear Logic in Computer Science. Cambridge University Press, Cambridge (2004)
13. Hickmott, S., Rintanen, J., Thiebaux, S., White, L.: Planning via Petri Net Unfolding. In: Proceedings of IJCAI, pp. 1904–1911 (2007)
14. Hodas, J.: Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation. Ph.D. Dissertation, University of Pennsylvania, Department of Computer and Information Science (1994)
15. Jacopin, E.: Classical AI Planning as Theorem Proving: The Case of a Fragment of Linear Logic. In: Proccedings of AAAI, pp. 62–66 (1993)
16. Kautz, H.A., Selman, B.: Planning as Satisfiability. In: Proccedings of ECAI, pp. 359–363 (1992)
17. Kautz, H.A., Selman, B., Hoffmann, J.: SatPlan: Planning as Satisfiability. In: Proccedings of 5th IPC (2006)
18. Korf, R.: Macro-operators: A weak method for learning. Artificial Intelligence 26(1), 35–77 (1985)
19. Küngas, P.: Analysing AI planning problems in linear logic – A partial deduction approach. In: Bazzan, A.L.C., Labidi, S. (eds.) SBIA 2004. LNCS, vol. 3171, pp. 52–61. Springer, Heidelberg (2004)
20. Küngas, P.: Linear logic for domain-independent ai planning. In: Proceedings of Doctoral Consorcium ICAPS (2003)
21. Lincoln, P.: Linear logic. In: Proceedings of SIGACT (1992)
22. Masseron, M., Tollu, C., Vauzeilles, J.: Generating plans in linear logic i-ii. Theoretical Computer Science 113, 349–375 (1993)
23. Oliet, N.M., Meseguer, J.: From petri nets to linear logic. In: Dybjer, P., Pitts, A.M., Pitt, D.H., Poigné, A., Rydeheard, D.E. (eds.) Category Theory and Computer Science. LNCS, vol. 389. Springer, Heidelberg (1989)
24. Tamura, N.: User's guide of a linear logic theorem prover (llprover) Technical report, Kobe University, Japan (1998)
25. Tanabe, M.: Timed petri nets and temporal linear logic. In: Proceedings of Application and Theory of Petri Nets, pp. 156–174 (1997)
26. Thielsher, M.: FLUX: A logic programming method for reasoning agents. Theory and Practice of Logic Programming 5(4-5), 533–565 (2005)

Constraint-Based Timetabling System for the German University in Cairo

Slim Abdennadher, Mohamed Aly, and Marlien Edward

German University in Cairo, Department of Computer Science,
5th Settlement New Cairo City, Cairo, Egypt
slim.abdennadher@guc.edu.eg, mohamed.aly@student.guc.edu.eg,
marlien.nekhela@student.guc.edu.eg
http://www.cs.guc.edu.eg

Abstract. Constraint programming is mainly used to solve combinatorial problems such as scheduling and allocation, which are of vital importance to modern business. In this paper, we introduce the course and examination timetabling problems at the German University in Cairo. As manual generation of schedules is time-consuming and inconvenient, we show how these problems can be modeled as constraint satisfaction problems and can be solved using SICStus Prolog with its powerful constraint library.

1 Introduction

Timetabling problems are real life combinatorial problems concerned with scheduling a certain number of events within a specific time frame. If solved manually, timetabling problems are extremely hard and time consuming and hence the need to develop tools for their automatic generation [6].

The German University in Cairo (GUC) started in Winter 2003. Each year the number of students increases and the timetabling problem whether for courses or examinations becomes more difficult. For the past four years, schedules for both courses and exams have been generated manually. The process involves attempting to assign courses/exams to time slots and rooms while satisfying a certain number of constraints. Processed manually the course timetabling took on average one month, while the examination timetabling two weeks. With the growing number of students, the GUC timetabling problems have become too complex and time-consuming to be solved manually.

In this paper, we will address the problems of the course and examination timetabling at the GUC. The course timetabling aims at generating a schedule which is personalized for all students in the different faculties at the university. This task should be done once every semester. The goal of examination timetabling is to produce a personalized examination schedule for all students enrolled in the GUC. It is generated three times per semester. Once for the midterm examinations, which occur in the middle of the semester and are distributed over a one week period; once during the final examinations, which occur at the end of the semester and are distributed over a two weeks period; and one

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 69–81, 2009.

last time for the make up examinations which occur after the end of the semester and are distributed over a two weeks period.

Several approaches have been used to solve such combinatorial problems [8], e.g., using an evolutionary search procedure such as ant colony optimization [3], or using a combination of constraint programming and local search techniques [7], or using graph coloring approach [2].

In this paper, we present an ongoing research to develop a constraint-based system to solve both of the course and examination timetabling problems at the GUC. Constraint programming is a problem-solving technique that works by incorporating constraints into a programming environment [1]. Constraints are relations which specify the domain of solutions by forbidding combinations of values. Constraint programming draws on methods from artificial intelligence, logic programming, and operations research. It has been successfully applied in a number of fields such as scheduling, computational linguistics, and computational biology.

The paper is organized as follows. The next section introduces the GUC examination and course timetabling problem and their requirements. Then we show how the problem can be modeled as a constraint satisfaction problem (CSP) and how we implemented the different constraints in SICStus Prolog. Finally, we conclude with a summary and future work.

2 Problem Description and Requirements

The GUC consists of four faculties, namely the Faculty of Engineering, the Faculty of Pharmacy and Biotechnology, the Faculty of Management, and the Faculty of Applied Arts. Currently, at the GUC, there are 140 courses offered and 5500 students registered for which course and examination timetables should be generated. There are 300 staff members available for teaching and invigilating as well as 1300 seating resources. Each faculty offers a set of majors. For every major, there is a set of associated courses. Courses are either compulsory or elective. Faculties do not have separate buildings, therefore all courses and all exams from all faculties should be scheduled taking into consideration shared room resources.

2.1 Course Timetabling

Students registered to a major are distributed among groups for lectures (lecture groups) and groups for tutorials or labs (study groups). A study group consists of maximum 25 students. Students registered to study groups are distributed among language groups. In each semester study groups are assigned to courses according to their corresponding curricula and semester. Due to the large number of students in a faculty and lecture hall capacities, all study groups cannot attend the same lecture at the same time. Therefore, groups of study groups are usually assigned to more than one lecture group. For example, if there are 27 groups studying Mathematics, then 3 lecture groups will be formed.

The timetable at the GUC spans a week starting from Saturday to Thursday. A day is divided to five time slots, where a time slot corresponds to 90 minutes. An event can take place in a time slot. This can be either a lecture, exercise, or practical session and it is given by either a lecturer or a teaching assistant. Usually, lectures are given by lecturers and exercise and practical sessions are given by teaching assistants. In normal cases, lectures take place at lecture halls, exercise sessions at exercise rooms and practical sessions take place in specialized laboratories depending on the requirements of a course. In summary, an event is given by a lecturer or an assistant during a time slot in a day to a specific group using a specific room resource. This relationship is represented by a timetable for all events provided that hard constraints are not violated. Hard constraints are constraints that cannot be violated and which are considered to be of great necessity to the university operation. These constraints can be also added by some lecturers like part timers or visiting professors. Hard constraints will be outlined and discussed later in this section. The timetable tries to satisfy other constraints which are not very important or critical. Such constraints are known as soft constraints that should be satisfied but maybe violated. For example, these constraints can come in form of wishes from various academic staff.

Courses have specific credit hours. The credit hours are distributed on lecture, exercise, or practical sessions. Every two credit hours correspond to a teaching hour. A teaching hour corresponds to a time slot on the timetable. Sessions with one credit hour corresponds to a biweekly session. These sessions are usually scheduled to take place with other biweekly sessions if possible. For example a course with 3 lecture credit hours, 4 exercise credit hour and 0 practical credit hours can be interpreted as having two lectures per week one being scheduled weekly and the other is biweekly, two exercise sessions per week and no practical sessions. Lecture groups and study groups take the corresponding number of sessions per week. Courses might have hard constraints. Some courses require specialized laboratories or rooms for their practical or exercise sessions. For example, for some language courses a special laboratory with audio and video equipment is required. The availability of required room resources must be taken into consideration while scheduling. Some lecturers have specific requirements on session precedences. For example, in a computer science introductory course a lecturer might want to schedule exercise sessions before practical sessions.

Furthermore, some constraints should be taken into consideration to improve the quality of education. An important constraint is that no lectures should be scheduled in the last slot of any day. Another constraint requires that a certain day should have an activity slot for students, and a slot where all university academics can meet. For those slots no sessions should be scheduled. A study group should avoid spending a full day at the university. In other words, the maximum number of slots that a group can attend per day is 4. Therefore, if a group starts its day on the first slot, then it should end its day at most on the fourth slot, and if it starts its day on the second slot, then it should end its day at most on the fifth slot. Furthermore, The number of days off of a group depends on the total number of credit hours a group has. For example, if a group has a total of 24 course credit hours in a semester, the total number of slots needed would be $\frac{24}{2} = 12$ slots, which corresponds to 3 study days, then a group may have $6 - 3 = 3$ days off in a week.

A certain number of academics are assigned to a course at the beginning of a semester. Teaching assistants are assigned to one course at a time. For courses involving practical and exercise sessions, a teaching assistant can be assigned to both an exercise and a practical session or one of them. This should be taken into consideration when scheduling to avoid a possible clash. Furthermore, the total numbers of teaching assistants assigned to a session should not exceed the maximum number of assistants assigned to the corresponding course at any time. A lecturer can be assigned to more than one course. This should be considered when scheduling in order to avoid a possible overlap. Academics assigned to courses have a total number of working and research hours per day and days off that need to be considered when scheduling. Courses should be scheduled in a manner such that the number of session hours given by an academic should not exceed his or her teaching load per day taking into consideration days off. For some courses, part-time lecturers are assigned. Those lecturers impose hard constraints most of the time. They might have special timing requirements or room requirements. These constraints are considered of a high priority and must be taken into consideration.

Another problem that resides when scheduling is the GUC transportation problem. The GUC provides bus transportation services for both students and academics. Two bus rounds take place in the beginning of the day to transport students and academics from various meeting points in Cairo. Other bus rounds take place, one after the fourth slot and the other after the fifth slot to transport students and academics back home. The first round delivers students just before the first slot and the second round delivers students just before the second slot. The number of students coming each day at the GUC by bus should be as balanced as possible in a way such that no limitation as possible is encountered with bus resources.

2.2 Examination Timetabling

The GUC examination timetabling problem aims at producing examination schedules for all courses given in a particular term.

The examination slots at the GUC differ from the midterms to the final examinations. Usually the midterm examinations are distributed over a one week period where the days are divided into four examination slots and the duration of each examination slot is 2 hours. The final examination period consist of two weeks with three examination slots per day where each examination slot is 3 hours.

The GUC examination timetabling problem consists of a set of courses and a set of students that need to be scheduled within limited resources of time and space. The task is to assign the student groups to examination slots and thus produce examination schedules for the midterm and final examinations. Since scheduling of the student groups from the different faculties is interdependent as time, space and specific courses are shared, schedules for the different faculties can not be produced separately and thus a single schedule for the whole university must be generated.

The GUC examination timetabling problem can be performed in two phases. In the first phase, the different course examinations are assigned to time slots while taking into account only the total number of students enrolled in the courses and the total number of available seats. In the second phase, after producing the examination schedule for all student groups, distribution of students among the rooms is done. In this paper, we will only model the first phase as a constraint satisfaction problem and solve it using constraint programming. The second phase does not require constraint techniques. A simple algorithm was designed and implemented using Java to distribute the students among the examination rooms and seats. In both phases, the different constraints that are imposed by the regulations of the GUC Examination Committee have to be taken into account. Dividing the examination scheduling problem into two parts reduces the complexity since the number of constraints that must be satisfied per phase is decreased. This division is feasible since the total number of seats is considered in the course scheduling (first) phase.

When assigning an examination to a certain time slot, several constraints must be considered.

Time Clash Constraint: A student group cannot have more than one examination at the same time.

Semester Clash Constraint: Student groups from the same major but in different semesters cannot have examinations at the same time. This is needed to enable students who are re-taking a course from a previous semester due to poor performance to sit for the course examination.

Core Exam Constraint: A student group cannot take more than one core examination per day.

Exam Restriction Constraint: A student group cannot have more than a certain number of examinations per day.

Difficulty Constraint: A student group cannot have two difficult examinations in two consecutive days (difficulty rating for a course is assigned by the course lecturer and students).

Capacity Constraint: The total number of students sitting for an examination in a certain time slot cannot exceed a predefined limit. This constraint ensures that the total number of students sitting for an examination at a certain time slot do not exceed the total number of available seats in the university campus.

Pre-assignment Constraint: An examination should be scheduled in a specific time slot.

Unavailability Constraint: An examination should not be scheduled in a specific time slot.

3 Modeling the Timetabling Problem as a CSP

Constraint satisfaction problems have been a subject of research in Artificial intelligence for many years [9]. A constraint satisfaction problem (CSP) is defined by a set of variables each having a specific domain and a set of constraints each involving a set of variables and restricting the values that the variables can take simultaneously. The solution to a CSP is an assignment that maps every variable to a value. To implement the problem, we have chosen the constraint logic programming (CLP) paradigm. This paradigm has been successful in tackling CSPs as it extends the logic paradigm by supporting the propagation of constraints for a specific domain providing an efficient implementation of computationally expensive procedures [4]. We have used the finite domain constraint library of SICStus Prolog (clpfd) to model the problem. In the following, the different types of constraints are described [5]:

Domain Constraints: Domain constraints are used to restrict the range of values variables can have. The constraint `domain` is used to restrict the domain of a variable or a list of variables. For example, the constraint `domain([s1,s2],5,25)` restricts the domain of `s1` and `s2` to be between 5 and 25.

Arithmetic Constraints: Many arithmetic constraints are defined in SICStus Prolog constraint library. Arithmetic constraints are in the form

```
Expr RelOp Expr
```

where `Expr` generally corresponds to a variable or a normal arithmetic expression, and `RelOp` is an arithmetic constraint operator like `#=`, `#\=`, `#<`, `#>`, `#>=`, and `#=<`. For example, `X#=Y` constraints the variables `X` and `Y` to be equal to each other, and `X+Y#>=3` constraints that the sum of `X` and `Y` should be greater than or equal to `3`.

Propositional Constraints: Propositional constraints enable the expressions of logical constraints. Two of the propositional constraints are the conjunction `X#/\Y` and disjunction `X#\/Y`. These constraints represent the conjunction and the disjunction of two constraints. For example, adding the constraint `X#=Y #\/ X#=Z` constrains that `X` can either be equal to `Y` or `Z`.

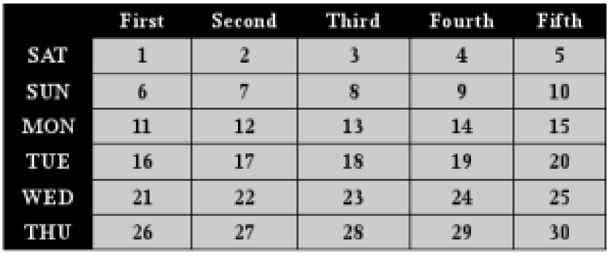
Combinatorial Constraints: Combinatorial constraints vary in form. Two of the widely used constraints are `all_different/1` and `count/4`. The input for the global constraint `all_different` is a list. The constraint ensures that all elements of the list are distinct. For example if the input list is a list of sessions from `s1..sN`, then the constraint will be added as `all_different([s1,s2,...,sN])`. Another combinatorial constraint used is `count/4`. The constraint is of the form `count(Val,List,RelOp,Count)`, where `Val` is a number, `List` is a list of variables, `RelOp` is an arithmetic constraint operator, and `Count` is a number. The constraint is satisfied if `N` is the number of elements of `List` that are equal to `Val` and `N RelOp Count`. For example, `count(1,[A,B,C],#<,2)` constraints that the total number of elements in the list `[A,B,C]` having the value `1` should be less than `2`.

Other combinatorial constraints used were the `serialized` and `cumulative`. The constraint `serialized/2` constrains x tasks, each with a starting time and duration so that no tasks overlap. The constraint `cumulative` consists of the following parameters: `Starts`, `Durations`, `Resources`, and `Limit`:

- `Starts = [S1,...,Sn]` is a list with the starting time of `n` tasks
- `Durations = [D1,...,Dn]` is a list with the durations of of `n` tasks
- `Resource = [R1,...,Rn]` is the amount of resources available for each task
- `Limit = [L1,...,Ln]` is the list with the maximum resource limit each task could use.

3.1 Course Timetabling Model

In our model, we have 5 slots in a day and 6 days in a week, our domain is chosen to be from 1 to 30 where each number corresponds to a slot. For example, slot 1 would correspond to the first slot on Saturday, and slot 23 would correspond to the third slot on Wednesday. Consequently, slots from 1 to 5 correspond to Saturday, 6 to 10 correspond to Sunday, 11 to 15 correspond to Monday, and so on. Figure 1 visualizes the timetable.



	First	Second	Third	Fourth	Fifth
SAT	1	2	3	4	5
SUN	6	7	8	9	10
MON	11	12	13	14	15
TUE	16	17	18	19	20
WED	21	22	23	24	25
THU	26	27	28	29	30

Fig. 1. CSP Model Timetable

At each slot a group attends an instance of a course (lecture, tutorial, or practical session). We define this relationship by the tuple

$(COURSE, GROUP, TYPE, SEQ, BI)$,

where $COURSE$ is a unique identifier of a certain course, $GROUP$ is a unique identifier of a certain group, $TYPE$ declares the course instance (LECT for a lecture session, EXER for an exercise session, and PRAC for a practical session), SEQ enumerates the instance number (for example if two lectures are offered within the week, then two sequences are defined, namely 1 and 2), and BI is a boolean value that defines if an instance should be scheduled as biweekly session. We define our variables to the problem by the tuple ($x_n = (COURSE, GROUP, TYPE, SEQ, BI)$) and the associated domain for each variable is a value from 1 to 30 ($D = \{1, ..., 30\}$).

Considering the relationship represented by the tuple, our variables correspond to lecture, exercise, and practical sessions taken by groups. The model expresses the problem of finding a slot for every course instance associated with a group such that constraints on the variables are not violated. In the next section we describe how we can model our set of constraints. The constraints will be applied until a solution is found. A solution describes when every session for each group would be scheduled taking into consideration the constraints of room, lecturers, and teaching assistant resources. The following describes the real constraints and how we used the SICStus clpfd library to model them:

Domain of Sessions is from 1 to 30: To restrict the domain of the sessions, we use the global constraint `domain`. In order to constraint the list `L` of all variables, we add the constraint `domain(L,1,30)`.

Group Session Overlap: The sessions of a study group should not overlap; i.e. a study group cannot attend two different courses at the same time. The global constraint `all_different` is used to model constraints on sessions where no overlap should be encountered. For example, if sessions `s1,s2,s3` belong to a group, then we add a constraint `all_different([s1,s2,s3])`.

Lecturer giving more than a course: Academic members who teach more than one course should not have their sessions overlapped while scheduling. To ensure this, we add an `all_different` constraint to constrain the sessions given by an academic member.

Full Day Prevention Constraint: To avoid bringing a group for a full day, then their associated sessions in a day should not be scheduled either in the first slot or in the last slot. To model this we use the inequality constraint with the conjunction `#/\` and the disjunction `#\/` constraints to exclude the first session and the last session of each day for every group. For example, if the following sessions `s1, s2, s3` are for a group and we would like to constraint the first day (slots 1,2,3,4,5), we add the constraint

```
(s1#\=1 #/\ s2 #\= 1 #/\s3 #\= 1)#\/
 (s1 #\= 5 #/\ s2 #\=5 #/\ s3 #\=5).
```

Course Precedence Requirements: For all courses that have special precedence requirements on sessions (i.e. if an exercise session must be scheduled before a practical session), we use the less than constraint `#<`. For example, if `s2` is before `s1` we add the constraint `s2#<s1`.

Part-Time Lecturer Constraints: If a part-time lecturer selects a certain slot to give a session, then all session variables given by this lecturer will be constrained, using the equality constraint, `#=`, to the required slot number. For example, consider a part timer giving the session `s1`, and can only come on the third and the fourth slots on Saturday (slots correspond to 3 and 4). We add the constraint `s1#=4 #\/ s1#=5`.

Lecturer Holiday Constraint: In order to express a lecturer holiday constraint and avoiding scheduling any courses taught by him on that day, we constrain all sessions given by the lecturer not to have the values of the day using the inequality constraint `#\=` with the disjunction constraint. For example, if a lecturer giving the session `s1` is off on Saturday, then we add the constraint `s1#\=1 #/\ s1#\=2 ... #/\ s1#\=5`.

Activity Slots, Free Sessions and Holidays: Slots that are required to remain empty with no scheduled sessions (as for the student activities slot) can be easily modeled by using the combinatorial constraint `count`. This is achieved by constraining the count of all variables in a list having the slot value is equal to zero. For example, if sessions `s1...sn` are not to be scheduled on the first session on Sunday, then we add the constraint `count(6,[s1...sn],#=,0)`.

Room Resources Constraint: In order to avoid scheduling exercise sessions in a slot more than the number of rooms available we constraint all the exercise sessions for every slot to be less than or equal the maximum number of available rooms using the `count` constraint with the arithmetic constraint operator `#=<`. For example if we cannot schedule more than 50 exercises per slot for sessions `s1,...,sN` we simply add `count(Slot,[s1,s2..sN],#=<,50)`, where `Slot` indicates the slot number (we have to define this constraint for all slots).

Teaching Assistant Resources: The `count` constraint is also used to limit the number of sessions scheduled per slot with the number of available teaching assistants assigned to the course or with the number of available room resources for a specific session (like a practical session).

Number of Lecture Sessions in a Day: We use the `count` constraint to constraint the total number of lectures per day. In order to model this we map all sessions to their corresponding day. This is done by subtracting 1 from the slot value and dividing by 5. By that we end with a value between 0 and 5 indicating the day. For example, for Saturday, the third slot has the value 3. Subtracting one from the value results in 2, dividing by 5 gives 0 which indicates day 0 (Saturday). As another example, for Monday, the second slot has the value 12. Performing the above operations, we get the value 2 which indicates the third day (Monday). We constraint the variables after applying the indicated operations (subtraction of one followed by division by 5) with the count constraint such that we have at most two lectures per day. For example, if lecture sessions `s1...sn` are taken by a group, we add the constraint `count(0,[(s1-1)/5,(s2-1)/5 ,...,(sn-1)/5],#=<,2)` for Saturday, and similarly for the rest of the week (Saturday is 0, Sunday is 1, Monday is 2, and so forth).

3.2 Examination Timetabling Model

In modeling the GUC examination timetabling problem as a CSP we use the following notation:

- n is the number of courses to be scheduled,
- $D = \{1, \ldots, m\}$ indicates a set of m time slots,
- $T = \{T_1, \ldots, T_n\}$ represents the time slot variables for all examinations, where T_i indicates the time slot assigned to an examination i,
- l is the number of all student groups in the university,
- The domain of every variable T_i is between 1 and m,

where m is the number of available time slots and can be calculated by multiplying the number of days of the examination period by the number of slots per day. For example, if the examination period consists of 6 days and every day consists of 3 time slots, then the available number of time slots is 18.

Different constraints are applied on T_i eliminating the different values each variable can take and thus reducing the domains of every variable to improve the efficiency of the search.

Now we describe the requirements of our problem in terms of SICStus Prolog using the constraint solver over finite domains (clpfd).

Time clash constraint stating that no student can have more than one exam at the same time slot can be enforced using the `all_different/1` global constraint. We have to constrain the l sets of variables representing the courses for every group of students to be different. Let `Tgroup = [T1,...,Tj]` be a list of variables that represent examination time slots of a specific group. Applying the `all_different/1` constraint on this list would ensure that all exams for that group are held in different time slots. Note that `Tgroup` is a subset of `T`.

In order to abide to the semester clash constraint, we have to take into consideration that all examinations of the same major should be scheduled at different time slots. Let `Y` be the number of courses of a certain major. `Tmajor = [T1,...,TY]` is a list of variables representing examination time slots for these courses. Applying the `all_different/1` constraint on these variables will ensure that all these exams are assigned to different time slots. The difficulty constraint has to be fulfilled in order to produce a valid examination schedule. We cannot schedule two difficult exams consecutively. Knowing the difficulty rating of every course, all time slot variables for difficult examinations can be known. For every two variables representing difficult examinations of a student group an inequality constraint ($\# \leq$) is imposed on their difference. This difference has to be greater than a certain number.

Enforcing the core exam constraint produces an acceptable examination schedule. A student cannot take more than one core exam per day. We can impose this constraint using the `serialized/2` global constraint. In our case, if we add to the duration of every core exam the number of slots per day, we can get a gap of at least one day between every two core examinations. Let `Z` be the number of core courses of a certain student group and `Tcore = [T1,...,TZ]` be a list variables that represent the time slots for these courses. Applying the following constraint

```
serialized(Tcore,[Gaps_added_durations])
```

will ensure that the gaps between core examinations is at least one day.

Exam restriction constraint limits the number of examinations held per day for every student to be less than a certain number. The variables that represent the examination time slots for every student group are known. The global constraint `count/4` can be used to limit the number of variables assigned to the same day to be less than a certain number. Let `x` be the number of courses assigned to a specific student group, `T = [T1,...,Tx]` is a list of variables representing the day of every examination slot and 1≤`Y`≤`m`, then `count(Y, T, #<, 3)` should be applied `m` times for every group of students constraining the number of exams held on every day to be less than 3.

In order to enforce the capacity constraint, the number of all students attending an examination at a certain time slot should not exceed a certain limit: The `cumulative/4` global constraint can be used to assign a number of examinations to the same time slot constraining the number of students to be less than the specified limit.

```
cumulative(Starts, Durations, Capacities, limit)
```

Here `Starts` is a list of variables representing all courses, `Durations` is a list of all examination durations, `Capacities` is a list of the number of students enrolled in every course, and `limit` is the number of seats available for examination.

Preassignment and unavailability constraints that restrict an exam to be scheduled in a given time slot or should not be scheduled in such time slot can be imposed using the equality and the inequality constraints. For example if a course `Ti` has to be assigned to time slot `j` then the constraint `Ti #= j` is added.

4 Implementation and Testing

Our system is composed of four components.

- A relational database that is used to store all needed information about courses, students and rooms.
- A Java based engine that is used to generate the CLP code and solves the second phase of the problem by assigning courses and students to examination rooms.
- A Prolog engine that is responsible for running the generated CLP code and getting a solution for the first phase of the problem by assigning a time slot to every examination.
- A graphical user interface implemented in Java that is used to display the results of both phases.

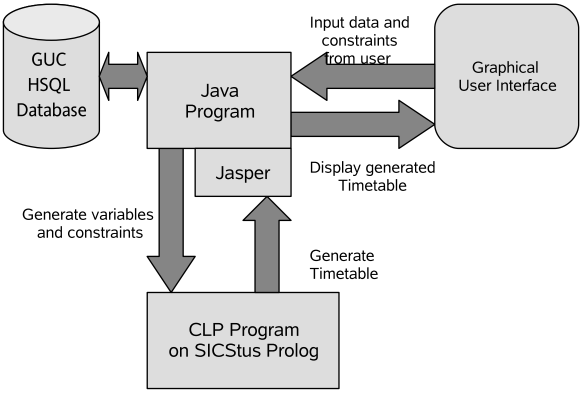

GUC HSQL Database
Input data and constraints from user
Java Program
Graphical User Interface
Jasper
Display generated Timetable
Generate variables and constraints
Generate Timetable
CLP Program on SICStus Prolog

Fig. 2. System Architecture

Figure 2 shows our system architecture. The relational database was used to store all the needed information about the input data such as courses, student groups and rooms as well as the output data such as the exams with their examination time slots and examination rooms. Relations between the different tables of the database are maintained in order to specify the courses of every student group and the rooms in which every exam of a course is held. An instance of an examination is expressed by a pair `exam(C,T)`, where `C` is a unique course identifier and `T` is the examination time slot assigned to that course. Instances of examinations of the different courses are the inputs to our problem.

In general, a CLP program consists of three basic parts. We begin by associating domains to the different variables we have, then the different constraints of the problem are specified, and finally a labeling strategy is defined to look for a feasible solution via a backtrack search or via a branch-and-bound search. Labeling, which is used to assign values to variables, is needed since constraint solving is in general incomplete. We employed the first-fail strategy to select the course to be scheduled and a domain splitting strategy to select a period from the domain of the selected course. First-fail selects one of the most-constrained variables, i.e. one of the variables with the smallest domain. Domain splitting creates a choice between `X #=< M` and `X #> M`, where `M` is the midpoint of the domain of `X`. This approach yielded a good first solution to our problem. The input to a CLP program is usually in the form of a list. Constraints are expressed using the predefined constraints implemented in the constraint solver over finite domains `clpfd`.

Initially, we query the database using Java to obtain all the information needed to be able to produce a valid assignment. The CLP code is generated such that it restricts the solutions using the constraints already chosen by the user through the graphical interface. The input to our problem is a list of timetable instances. While generating that list we constrain the values of the time slot variables using the membership global constraint `domain/3`. After generating the list of instances, different constraints are applied to the variables eliminating values of their domains until finally a solution is reached.

After generating the SICStus Prolog code, Jasper[1] is used to consult the program and parse the values of the variables that represent the time slots for the different examinations. These values with their associated examination instances are stored in the database. A graphical user interface is then used to display the generated timetable.

Our system was tested on realistic data and constraints for the generation of the course and examination timetable for the Winter term 2007-2008. Our testing environment was a Microsoft Windows Vista Ultimate running on a 32-bit processor at 1.73 GHz with 1 GB of RAM. Solutions were generated in a few seconds. For examination timetabling, the test data consisted of 140 courses, 5500 students, and 1300 available seats for them. For the course timetabling, we have tested the system on three main different study groups majoring in their second semester with a total of 473 course instance variables (corresponding to

[1] A Prolog interface to the java virtual machine.

lecture, exercise, and practical session) to be scheduled and we generated a good solution within a few seconds.

5 Conclusion and Future Work

In this paper, the GUC course and examination timetabling problems have been discussed and possible solutions for both problems using constraint logic programming in SICStus Prolog have been presented. The time taken to implement, debug and test the software was about 14 weeks. Due to the declarativity of our approach, our tool can be easily maintained and modified to accommodate new requirements and additional constraints.

One direction of future work is to use the constraint programming technology presented in this paper within a local search framework to optimize and repair the solutions found. This will lead to a hybrid method which was proved to be a powerful method to solve timetabling problems [7].

Another enhancement would be to integrate this system with the invigilation timetabling system implemented at the GUC. This system schedules invigilators such as professors, doctors and teaching assistant to proctor the scheduled examinations.

References

1. Abdennadher, S., Frühwirth, T.: Essentials of Constraint Programming. Springer, Heidelberg (2002)
2. Petrovic, S., Burke, E., Dror, M., Qu, R.: Hybrid graph heuristics within a hyper-heuristic approach to exam timetabling problems. In: The Next Wave in Computing, Optimization, and Decision Technologies (2005)
3. Eley, M.: Ant Algorithms for the Exam Timetabling Problem. In: Burke, E.K., Rudová, H. (eds.) PATAT 2007. LNCS, vol. 3867. Springer, Heidelberg (2007)
4. Fernández, A.J., Hill, P.M.: A comparative study of eight constraint programming languages over the boolean and finite domains. Constraints 5(3), 275–301 (2000)
5. Intelligent Systems Laboratory. SICStus Prolog User's Manual. Swedish Institute of Computer Science, PO Box 1263, SE-164 29 Kista, Sweden (April 2001)
6. McCollum, B., Ireland, N.: University timetabling: Bridging the gap between research and practice. In: Burke, E.K., Rudová, H. (eds.) PATAT 2006. LNCS, vol. 3867. Springer, Heidelberg (2007)
7. Merlot, L.T.G., Boland, N., Hughes, B.D., Stuckey, P.J.: A hybrid algorithm for the examination timetabling problem. In: Burke, E.K., Rudová, H. (eds.) PATAT 2007. LNCS, vol. 3867. Springer, Heidelberg (2007)
8. Schaerf, A.: A survey of automated timetabling. In: Centrum voor Wiskunde en Informatica (CWI), vol. 115, pp. 30–33 (1995) ISSN 0169-118X
9. Tsang, E.: Foundation of Constraint Satisfaction. Academic Press, London (1993)

Squash: A Tool for Analyzing, Tuning and Refactoring Relational Database Applications[⋆]

Andreas M. Boehm[1], Dietmar Seipel[2], Albert Sickmann[1], and Matthias Wetzka[2]

[1] University of Würzburg, Rudolf–Virchow–Center for Experimental Biomedicine
Versbacher Strasse 9, D–97078 Würzburg, Germany
ab@andiboehm.de, albert.sickmann@virchow.uni-wuerzburg.de

[2] University of Würzburg, Department of Computer Science
Am Hubland, D–97074 Würzburg, Germany
seipel@informatik.uni-wuerzburg.de, matthias.wetzka@web.de

Abstract. The *performance* of a large biological application of relational databases highly depends on the quality of the database schema design, the resulting structure of the tables, and the logical relations between them.

We have developed a tool named Squash (SQL Query Analyzer and Schema EnHancer) for visualizing, analyzing and refactoring database applications. Squash parses the SQL definition of the database schema and the queries into an XML representation called SquashML, which is then processed in SWI–PROLOG and the integrated XML *query* and *transformation* language FNQuery.

Squash comes with a set of *predefined methods* for tuning the database application according to the load profile, and with methods for proposing refactorings, such as index creation, partitioning, splitting, or further normalization of the database schema. SQL statements are adapted simultaneously upon modification of the schema. Moreover, the declarative Squash framework can be flexibly extended by *user–defined methods*.

1 Introduction

During productive use, enterprise–class databases undergo a lot of changes in order to keep up with ever–changing requirements. The growing space requirements and complexity of productive databases make the task of maintaining a good performance of the database query execution more and more complicated. Performance is highly dependent on the database schema design [11,13,20], and additionally a complex database schema is more prone to design errors.

Increasing the performance and the manageability of a database usually requires restructuring the database schema and has effects on the application code. In addition, tuning query execution is associated with adding secondary data structures such as indexes or horizontal partitioning [2,8,16]. Because applications depend on the database schema, its modification implies the adaption of the queries used in the application code. The tuning of a complex database is usually done by specially trained experts having good knowledge of database design and many years of experience with tuning

[⋆] This work was supported by the Deutsche Forschungsgemeinschaft (FZT 82).

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 82–98, 2009.

databases [18,24]. Due to the lot of possibilities, the optimization of a database is a very complex and time–consuming task, even for professional database experts. In the process of optimization, many characteristic values for the database need to be calculated and assessed. Special tuning tools can help the database administrator to focus on the important information and support complex database schema manipulations [18,24]. More sophisticated tools can even propose improved database configurations by using the database schema and a characteristic workload log as input [1,6,14].

During the last two decades, approaches towards self–tuning database management systems have been developed [7,10,18,27,29,30]. These approaches have inherent access to the most current load profile, which is acquired online by the database management system. But they are limited to changes in the database that are transparent to the application code. Thus, they lack the possiblity of automatically updating the underlying source code and are not capable of refactoring applications completely. The approach presented in this paper aims at refactoring the database schemas and the queries in the SQL code of the application at the same time.

The rest of the paper is organized as follows: Section 2 describes the representation of SQL create statements in SquashML and the basic concepts of the XML query and transformation language FNQuery. As an example, foreign key (FK) constraints in the database schema are detected and visualized. Section 3 summarizes the functionality of the Squash system and gives some further examples for the analysis of database schemas. Section 4 represents SQL query statements in SquashML and analyzes the relationship of join conditions in the queries and FK constraints in the database schema. Finally, Section 5 investigates database tuning by indexes and briefly reports about a case study that was conducted with Squash on a biological database application.

2 Management of SQL Data with PROLOG and FNQuery

We have developed an XML representation called SquashML for SQL database schema definitions and queries. The core of SquashML follows the SQL standard, but SquashML also allows for system–specific constructs of different SQL dialects; for example, some definitions and storage parameters from the Oracle database management system have been integrated as optional XML elements.

SquashML is able to represent schema objects, as well as database queries obtained from application code or from logging data. This format allows for easily processing and evaluating the database schema information. Currently, supported schema objects include table and index definitions. Other existing XML representations of databases like SQL/XML usually focus on representing the database contents, i.e. the tables, and not the schema definition itself [17]. SquashML was developed specifically to map only the database schema and queries, without the current contents of the database.

The management of the SquashML data is implemented using the declarative logic programming language PROLOG [9,25], the Squash parser is implemented in Perl. XML documents are represented using the *field notation* data structure [22], and the XML query and transformation language FNQuery [22,23] is used. This language resembles XQuery [5], but it is implemented in and fully interleaved with PROLOG. The usual axes of XPath are provided for selection and modification of XML documents.

Moreover, FNQuery embodies transformation features, which go beyond XSLT, and also update features.

The Squash system comes with a library of predefined methods, which are also accessible through a graphical user interface. It can be flexibly extended by plugging in further, used–defined methods. In the following, we will show how such methods can be defined using PROLOG and FNQuery on SquashML. As a first example we will show how to detect and visualize foreign key constraints in SQL create statements.

Representation of Create Statements in SquashML

Consider the following abbreviated and simplified statement from a biological database for Mascot$^{\text{TM}}$ [19] data:

```
CREATE TABLE USR_SEARCH.
    TBL_SEARCH_RESULTS_MASCOT_PEPTIDE (
  ID_SEARCH_RESULTS_MASCOT_PEPTIDE NUMBER DEFAULT -1
     NOT NULL primary key,
  ID_RES_SEARCH NOT NULL,
  ID_RES_SPECTRA NOT NULL,
  ID_RES_SEARCH_PEPTIDES NOT NULL,
  ID_DICT_PEPTIDE NOT NULL,
  foreign key (ID_DICT_PEPTIDE)
    references TBL_DICT_PEPTIDES,
  foreign key (ID_RES_SEARCH_PEPTIDES)
    references TBL_RES_SEARCH_PEPTIDES ... );
```

The Squash parser transforms this into SquashML; for simplicity we leave out some closing tags:

```
<create_table
  name="TBL_SEARCH_RESULTS_MASCOT_PEPTIDE"
  schema="USR_SEARCH">
  <relational_properties>
    <column_def
      name="ID_SEARCH_RESULTS_MASCOT_PEPTIDE"> ...
    <column_def name="ID_RES_SEARCH">
      <inline_constraint>
        <inline_null_constraint type="not_null" /> ...
    <column_def name="ID_RES_SPECTRA"> ...
    <column_def name="ID_RES_SEARCH_PEPTIDES"> ...
    <column_def name="ID_DICT_PEPTIDE"> ...
    <out_of_line_constraint name="...">
      <out_of_line_foreign_key_constraint>
        <column name="ID_DICT_PEPTIDE" />
        <references_clause
          object="TBL_DICT_PEPTIDES"/> ...
    <out_of_line_constraint name="...">
      <out_of_line_foreign_key_constraint>
        <column name="ID_RES_SEARCH_PEPTIDES" />
        <references_clause
```

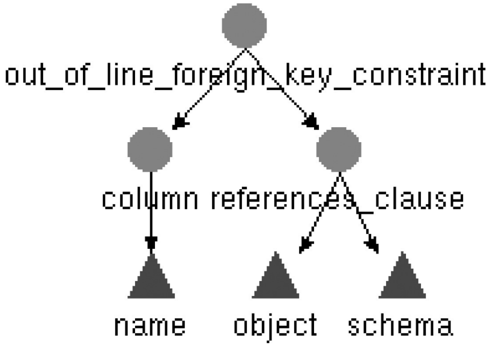


Fig. 1. Foreign Key Constraint in SquashML

```
          object="TBL_RES_SEARCH_PEPTIDES"/> ...
    </relational_properties>
  </create_table>
```

The structure of foreign key (FK) and primary key (PK) constraints [20] can be visualized using Squash, cf. Figure 1 and Figure 2, which helps to get an overview of the complex elements.

Processing of Create Statements in PROLOG and FNQuery

The following PROLOG rules determine an FK constraint and the PK of the referenced table from a SquashML database schema `Dbs`. The first PROLOG predicate selects an `out_of_line_foreign_key_constraint` within the create statement of a table `T1`, and then it selects the referencing column `A1` as well as the referenced table `T2`; subsequently, a call to the second PROLOG predicate determines the PK `A2` of the referenced table `T2`:

```
squashml_to_fk_constraint(Dbs, T1:A1=T2:A2) :-
   C := Dbs/create_table::[@name=T1]/_
      /out_of_line_foreign_key_constraint,
   A1 := C/column@name,
   T2 := C/references_clause@object,
   squashml_to_pk_constraint(Dbs, T2:A2).

squashml_to_pk_constraint(Dbs, T:A) :-
   _ := Dbs/create_table::[@name=T]/_/column_def::[@name=A]
      /inline_constraint/inline_primary_key_constraint.
```

In FNQuery, elements or attributes can be selected from XML elements using the binary infix predicate `:=`, which evaluates its second argument and returns the result in the first argument. E.g., the following is a selection expression:

```
C:= Dbs/create_table::[@name=T1]/_
   /out_of_line_foreign_key_constraint.
```

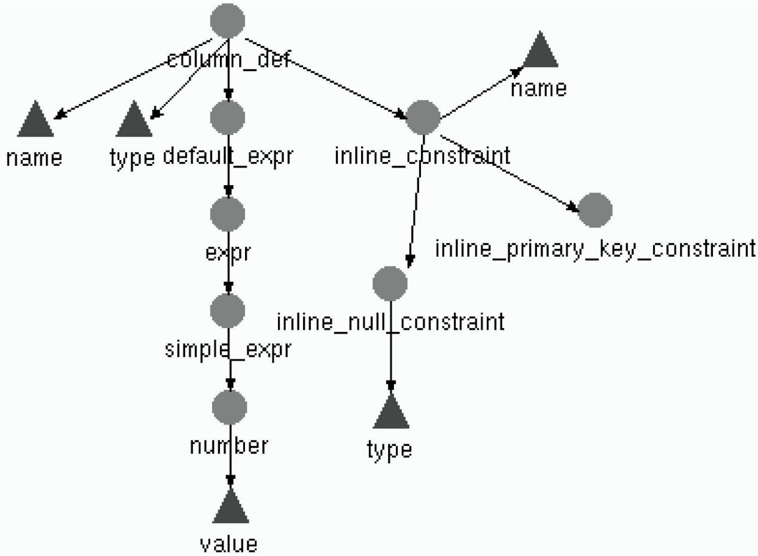


Fig. 2. Primary Key Constraint in SquashML

A location step in FNQuery for elements is of the form `/Ax::Nt` or `/Ax::Nt::Ps`, where `Ax` is an axis, `Nt` is a node test and `Ps` is a list of predicates. In this paper, we have only used the axes `child`, `descendant`, and `self`. For `child`, the location steps are abbreviated to `/Nt` and `/Nt::Ps`. E.g., `/create_table::[@name=T1]` selects a subelement with the tag `create_table`; the predicate `@name=T1` ensures that the selected element has an attribute `name` and selects the value `T1` of this attribute. A location step `/descendant::'*'` can be abbreviated to `/_`. For selecting an attribute, a location step `/attribute::A` is used, which can be abbreviated to `@A`.

There exist 18 FK constraints in the whole Mascot^{TM} part of the `resDB` database schema. Figure 3 visualizes the computed FK constraints of a fragment of the database; e.g., for the tables `A=TBL_SEARCH_RESULTS_MASCOT_PEPTIDE` and `B=TBL_RES_SEARCH_ PEPTIDES`, the FK constraint from the column `ID_RES_SEARCH_PEPTIDES` of `A` to the primary key column with the same name of `B` is shown as the rhombus labelled by `fk4`.

3 The Squash System

Squash is part of the DDK (DisLog Developers' Toolkit), a large SWI–PROLOG library [26], which can be loaded into applications on either Windows, UNIX or Linux. The DDK can be downloaded from `http://www1.informatik.uni-wuerzburg.de/database/DisLog/`.

The Squash system is divided into components for parsing, visualization, analysis, tuning, and refactoring:

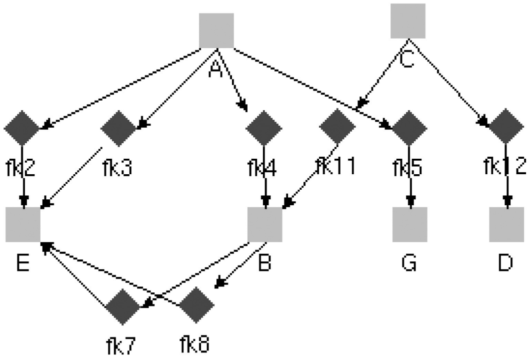


Fig. 3. Foreign Key Constraints in the Database

- The *visualization* and *analysis* components can help a database administrator to gain a quick overview of the most important characteristics of the database. Squash provides functions to view the different aspects of the database schema with visual markup of important properties and to analyze a database schema and a corresponding workload for semantic errors, flaws and inconsistencies.
- The *tuning* component accesses the data of the analysis component and uses heuristic algorithms to automatically optimize the database schema.
- Then, the *refactoring* component of Squash applies the suggested, potentially complex changes to the database schema. It can automatically propagate schema changes to the queries of the application.

Visualization

Comprehending the structure of a complex database schema just by reading the SQL create statements is a very demanding task, and design errors can be missed easily. Squash provides a number of different visualization methods for the database schema (cf. Figure 3) and the queries (cf. Figure 4). Complex select statements tend to include many tables in join operations. Therefore, Squash uses a graph representation for query visualization. If a select statement contains nested subqueries, then these queries can be included if desired.

Analysis

The analysis component of Squash performs multiple tasks. First, in addition to a graphical overview, more advanced Squash methods check the database schema and the queries for possible design flaws (anomalies) or semantic inconsistencies. Using this information, the database administrator can manually decide about optimization methods for the database schema. Secondly, when using the automatic tuning methods of Squash, these data are used as input for heuristic algorithms.

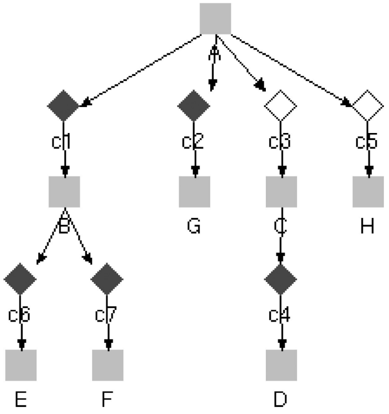


Fig. 4. Join Conditions in a Query

Design Anomalies. Especially within queries, semantic errors are often introduced by inexperienced database programmers [4,15]. The predefined methods of Squash include the detection of constant output columns, redundant output columns, redundant joins, incomplete column references, and join conditions lacking FK relationships (see Section 2).

The database schema can also be checked for design flaws. The available Squash methods include the detection of isolated schema parts, cyclic FK references (an implementation is given below), unused columns or tables, unique indexes that are defined as non–unique, indexes of low quality, datatype conflicts, anonymous PK or FK constraints, missing primary keys, and the detection and improvement of bad join orders.

```
squashml_to_isolated_schema_part(Dbs, Table) :-
   findall( T1-T2,
      squashml_to_fk_constraint(Dbs, T1:_=T2:_),
      Edges ),
   edges_to_connected_components(Edges, Components),
   member([Table], Components).

squashml_to_cyclic_fk_reference(Dbs, T1:A1=T2:A2) :-
   findall( V-W,
      squashml_to_fk_constraint(Dbs, V=W),
      Edges ),
   edges_to_cycle(Edges, T1:A1-T2:A2).
```

The edges of a graph are represented as pairs of the form `V-W`. A table is isolated in the database schema, if it forms a singleton connected component w.r.t. the FK references.

Database Characteristics and Schema Properties. The *space requirements* of the database tables is one of the most important pieces of information necessary for good database tuning. The analysis component of Squash uses the formulas available from the documentation of the database management system in oder to calculate the exact space requirements of a given table using the following predicate:

```
squashml_to_table_row_size(
      Dbs, Database:Table, Row_Size) :-
   squashml_to_table_columns(Dbs, Table, Columns),
   maplist( database_to_column_size(Database:Table),
      Columns, Sizes ),
   squash_column_sizes_to_row_size(Sizes, Row_Size).

squashml_to_table_columns(Dbs, Table, Columns) :-
   findall( Column,
      Column := Dbs/create_table::[@name=Table]
         /relational_properties/column_def@name,
      Columns ).

database_to_column_size(
      Database:Table, Column, Col_Size) :-
   Select = [
      use:[Database],
      select:[avg(vsize(Column))],
      from:[Table] ],
   database_select_execute(Select, [[Col_Size]]).
```

The predicate `database_to_column_size/3` accesses the underlying SQL database, and the predicate `squash_column_sizes_to_row_size/2` suitably aggregates the column sizes to the total size of the row.

Additionally, the analysis component collects schema properties and displays them. These functions are used for creating reports about the database schema and the queries.

Refactoring and Tuning

The refactoring component allows for manipulating the database schema as well as the corresponding application code based on the transformation and update features of FnQuery. Besides trivial manipulations such as adding or removing colums, the system also supports complex schema manipulations that affect other parts of the database schema and the queries, such as vertical partitioning and merging and splitting of tables.

For tuning, Squash is able to provide all queries in the workload with the best set of indexes according to a heuristic function (see Section 5), and Squash can generate appropriate optimizer hints.

4 Visualization and Analysis of SQL Queries

In the following we will show how to represent SQL queries in SquashML. Squash can analyse and visualize complex SQL query statements and relate them to the underlying database schema. As an example, we will show how we can find join conditions in queries that do not correspond to FK constraints in the database schema.

Representation of Select Statements in SquashML

E.g., the following SQL select statement from the Mascot^{TM} database joins 8 tables:

```
SELECT *
FROM TBL_SEARCH_RESULTS_MASCOT_PEPTIDE A,
   TBL_RES_SEARCH_PEPTIDES B,
   ... C, ... D, ... E, ... F, ... G, ... H
WHERE A.ID_DICT_PEPTIDE IN (
   SELECT B0.ID_PEPTIDE
   FROM TBL_RES_SEARCH_PEPTIDES B0
   WHERE B0.ID_RES_SEARCH = 2025
   GROUP BY ID_PEPTIDE
   HAVING COUNT(ID_RES_SEARCH_PEPTIDES) >=1 )
AND A.ID_RES_SEARCH = 2025
AND c1 AND c2 AND A.FLAG_DELETE = 0
AND c3 AND c6 (+) AND c7 (+)
AND c4 AND c5 AND E.LEN >= 6
AND A.SCORENORMALIZED >= 1.5
ORDER BY D.ID_SEQ_SEQUENZ, B.ACC_ID_CLS, ...;
```

It uses the following 7 join conditions:

```
c1: A.ID_RES_SEARCH_PEPTIDES = B.ID_RES_SEARCH_PEPTIDES
c2: A.ID_RES_SPECTRA = G.ID_RES_SPECTRA
c3: A.ID_RES_SEARCH_PEPTIDES = C.ID_PEPTIDE
c4: C.ID_SEQUENCE = D.ID_SEQ_SEQUENZ
c5: A.ID_RES_SEARCH = H.ID_RES_SEARCH
c6: B.ID_PEPTIDE = E.ID_DICT_PEPTIDE
c7: B.ID_PEPTIDE_MOD = F.ID_DICT_PEPTIDE
```

The query is parsed into the following SquashML element; again, we leave out some closing tags:

```
<select>
  <subquery id="subquery_1">
    <select_list>
      <expr>
        <simple_expr>
          <object table_view="A" column="IDRESSEARCH"/> ...
    <from>
      <table_reference>
        <query_table_reference>
          <query_table_expression>
            <simple_query_table_expression
              object="TBL_SEARCH_RESULTS_MASCOT_PEPTIDE"
              schema="USRSEARCH"/> ...
          <alias name="A"/> ...
    <where> ...
    <order_by> ...
</select>
```

The conditions in the WHERE part look like follows:

```
<condition>
  <simple_comparison_condition operator="=">
    <left_expr>
      <expr>
        <simple_expr>
          <object table_view="A"
            column="ID_RES_SEARCH_PEPTIDES"/> ...
    <right_expr> ...
          <object table_view="B"
            column="ID_RES_SEARCH_PEPTIDES"/> ...
</condition>
```

Processing of Select Statements in PROLOG and FNQuery

We wanted to know which of the join conditions of the query are valid FK constraints, cf. Figure 3, since other join conditions might be design errors. The following PROLOG rule selects a simple comparison condition with the operator "=", or an outer join condition:

```
squashml_to_join_condition(Dbs, T1:A1=T2:A2) :-
   S := Dbs/select,
   ( Condition := S/_
        /simple_comparison_condition::[@operator=(=)]
   ; Condition := S/_
        /outer_join_condition ),
   condition_to_pair(Condition, left_expr, T1:A1),
   condition_to_pair(Condition, right_expr, T2:A2).
```

Within the selected conditions, the following rule determines the left and the right table/column pair:

```
condition_to_pair(Condition, Tag, T:A) :-
   [T, A] := Condition/Tag/expr/simple_expr
      /object@[table_view, column].
```

The combined location step `@[table_view, column]` selects a list `[T, A]` of two attribute values. Now, the join conditions that are no valid FK constraints can be computed as follows:

```
squashml_to_unfounded_join_condition(Dbs, T1:A1=T2:A2) :-
   squashml_to_join_condition(Dbs, T1:A1=T2:A2),
   \+ squashml_to_fk_constraint(Dbs, T1:A1=T2:A2).
```

In Figure 4, the join conditions which are valid FK constraints are depicted as blue (dark) rhombs, whereas the others are coloured in white. E and F are aliases for the same table. It turned out, that the join condition c3 between A and C does not correspond to a valid FK constraint, but there exist two other FK constraints which link A and C through the table B, cf. Figure 3 (note that the labels of the constraints in the two figures don't correspond):

```
fk4:  A:ID_RES_SEARCH_PEPTIDES
        -> B:ID_RES_SEARCH_PEPTIDES
fk11: C:ID_PEPTIDE
        -> B:ID_RES_SEARCH_PEPTIDES
```

Also, the join condition c5 between A and H does not correspond to a valid FK constraint. In this example, both join conditions c3 and c5 are correct – and they are no design errors.

5 Database Tuning

The tuning component of Squash supports the proposal and the creation of indexes as well as of horizontal partitions. The system analyzes the database schema in conjunction with the workload and generates suitable refactorings. Many problems in database tuning, such as determining an optimal set of indexes, are known to be $\mathcal{NP}$–complete [21]. Since computing exact optimality is much too complex, heuristic algorithms are used for optimizing the database schema.

Index Declarations in Database Schemas

An already existing index can be found in the SquashML database schema Dbs using the following predicate squashml_index/4:

```
squashml_index(Dbs, Index, Table, Columns) :-
   setof( Column,
      ( table_index_column(Dbs, Index, Table, Column)
      ; primary_key_column(Dbs, Index, Table, Column) ),
      Columns ).

table_index_column(Dbs, Index, Table, Column) :-
   Column := Dbs/create_index::[@name=Index]
      /table_index_clause::[@table=Table]
      /index_expr/column@name,

primary_key_column(Dbs, Name, Table, Column) :-
   P := Dbs/create_table::[@name=Table]
      /relational_properties,
   ( Column := P/out_of_line_constraint::[/self::'*'=C]
        /out_of_line_primary_key_constraint/column@name
   ; C := P/column_def::[@name=Column]
        /inline_constraint::
           [/inline_primary_key_constraint] ),
   fn_get_optional_attribute(C@name, '', Name).
```

An index can be evaluated based on the heuristic function which is used in the following approach.

Index Proposal

The index proposal algorithm is able to propose and generate multi–column indexes (either unique or non–unique). The solution–space is reduced to a manageable size by a *heuristic multi–step approach* in combination with a scoring function.

In the first step, statistical data are collected from the database. This information is used to minimize the set of columns that contribute to an optimized attribute combination. In the second step, the workload is analyzed, and the quality of each column is calculated. All columns whose quality is below a threshold are removed from the set of candidate columns. Finally, the remaining columns are evaluated in detail, and a solution in the form of a sequence of columns is generated.

The algorithms have been developed without a special database management system in mind, but currently they are parameterized for application to Oracle.

The Method. Given a set $\mathcal{W}$ of SQL statements, the *workload*. For each single query $q \in \mathcal{W}$ and each table t that is accessed by q, we try to propose a suitable index on t. Let $V_{q,t}$ be the set of columns of t that are used in q. First, a subset $V'_{q,t} \subseteq V_{q,t}$ is calculated, which contains all columns of t that would be plausible as index columns to improve q (without considering any further information from the workload). Secondly, the best subset $V^*_{q,t} \subseteq V'_{q,t}$ according to a quality function $qual_{\mathcal{W},q,t}$ is determined. Thirdly, the algorithm proposes a permutation $\mathcal{V}^*_{q,t}$ of the elements of $V^*_{q,t}$ as an index on t to improve q. In the following, we describe the three steps in more detail.

Step 1. During the selection of $V'_{q,t}$, the algorithm retrieves information from the database schema and the workload. For each column $A \in V_{q,t}$, the quality $qual_t(A)$ is computed based on the following features: the selectivity and the data type of A, the table size of t, the change frequency of A, the number of occurences of A in SELECT/UPDATE statements (in the WHERE or the ORDER BY clause) or foreign keys. Only columns whose quality is above a certain threshold γ are considered further in the second step: $V'_{q,t} = \{ A \in V_{q,t} \mid qual_t(A) \geq \gamma \}$.

Step 2. For selecting the *best subset* $V^*_{q,t} \subseteq V'_{q,t}$, a quality function $qual_{\mathcal{W},q,t}$ is defined, which consists of a part depending on t and V and a part that depends on $\mathcal{W}$, q, and V. First, we define $qual_t(V)$ as the average of the qualities $qual_t(A)$ of all columns $A \in V$. Then, the selectivity $sel_{\mathcal{W}}(V)$ is defined. The workload $\mathcal{W}$ is scanned for occurrences of the columns $A \in V$ within the conditions P of the queries. Each condition P is assigned a value $sel(P)$ between 0 and 1 representing the *selectivity* of the predicate symbol of P; range predicates (such as $\leq$, $\geq$, $<$ and $>$) have low values, whereas equality predicates (such as $=$, LIKE, and IN) have higher values. If a column occurs in more than one condition P, then the values are multiplied to reflect the further restriction of the result set. Let $\mathcal{P}_{\mathcal{W}}(A)$ be the set of all conditions P in the whole workload $\mathcal{W}$, such that A occurs in P:

$$sel_{\mathcal{W}}(V) = \prod_{A \in V} \prod_{P \in \mathcal{P}_{\mathcal{W}}(A)} sel(P).$$

The frequency $freq_{\mathcal{W}}(q)$ is the number of occurrences of a query $q \in \mathcal{W}$ in the workload $\mathcal{W}$. Finally, the quality of $V \subseteq V'_{q,t}$ is calculated as the product of the previously described measures; to compensate a possible high rating of index propsals with a small number of columns, the quotient of $|V|$ and $|V_{q,t}|$ is a further factor:

$$qual_{\mathcal{W},q,t}(V) = \frac{|V|}{|V_{q,t}|} \cdot sel_{\mathcal{W}}(V) \cdot freq_{\mathcal{W}}(q) \cdot qual_t(V).$$

Step 3. If the quality $qual_{\mathcal{W},q,t}(V^*_{q,t})$ of the best set $V^*_{q,t}$ is greater than a given threshold γ', then Squash proposes a permutation $\mathcal{V}^*_{q,t}$ of the columns in $V^*_{q,t}$ as an index on t. In a case study, a value of $\gamma' = 4$ has been found to produce good results for Oracle, and $\gamma = 9$ was used for computing $V'_{q,t}$. For producing the permutation $\mathcal{V}^*_{q,t}$, the algorithm sorts the columns in an order being most useful for the queries in the workload. Squash offers three different sorting methods. The first one orders the columns by decreasing selectivity. The second method sorts them according to the type of condition they are used in. The third method takes the reversed order in which the columns appear in the query. Which one produces the best results, depends on the query optimizer of the database management system. In a case study, sorting according to the WHERE clause was found to yield good results for Oracle.

Implementation of the Selectivity in Squash. As a part of the index proposal method, we show the following PROLOG predicate for computing the selectivity $sel_{\mathcal{W}}(V)$ of a set of columns of the table relative to the workload.

```
squashml_to_selectivity(Workload, T, V, Sel) :-
   findall( S,
      ( member(A, V),
        squashml_to_occurrence_of_column(
           Workload, T:A, Condition),
        selectivity(Condition, S) ),
      Sels ),
   product(Sels, Sel).

squashml_to_occurrence_of_column(
      Workload, T:A, Condition) :-
   ( S := Workload/select
   ; S := Workload/update ),
   Condition := S/_/simple_comparison_condition,
   ( condition_to_pair(Condition, left_expr, T:A)
   ; condition_to_pair(Condition, right_expr, T:A) ).
```

The second predicate above computes the conditions in the workload where a column occurs; i.e., on backtracking the set $\mathcal{P}_{\mathcal{W}}(A)$ is generated.

Case Study on Index Proposal. A case study was conducted using a data warehouse system designed for use in mass spectrometry–based proteomics, that provides support for large scale data evaluation and data mining. This system, which is composed of the two subsystems `seqDB` and `resDB` [3,31], was temporarily ported to Oracle for being analyzed by Squash. The complete database schema consists of 46 tables requiring about 68.5 GB disk space.

Some characteristic SQL queries of the data evaluation part `resDB`, which were obtained from an application log, were submitted to Squash for analysis in conjunction with the schema creation script. These queries perform the grouping of peptide results, that were obtained by Sequest™ [12] and Mascot™ [19], respectively, into protein identification results. For demonstration, the Sequest™ queries were previously tuned manually by an experienced database administrator, the Mascot™ queries were left completely untuned.

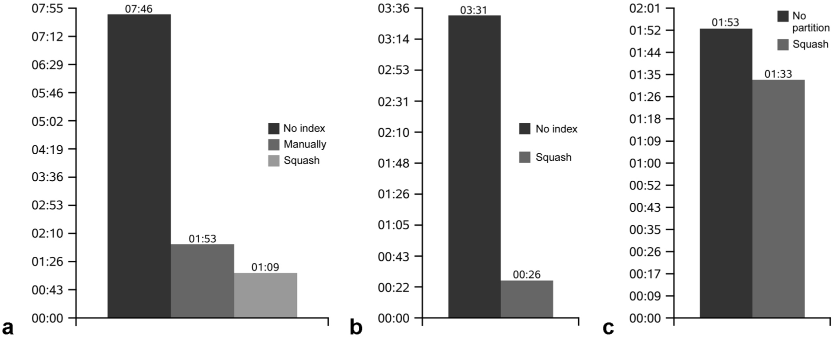


Fig. 5. Runtimes of Characteristic SQL Queries

The runtimes and the improvements resulting from the tuning, which were between 20% and 80%, are depicted in Figure 5.

- 5.a shows a Sequest™ query with indexes proposed by Squash. The duration was 7:46 min without any indexes. This could be reduced by manual tuning to 1:53 min, whereas tuning by Squash achieved a further reduction of about 49% down to 1:09 min, i.e. 14% of the original time.
- 5.b shows a Mascot™ query with indexes proposed by Squash. The execution time was 3:31 without any indexes. This was not tuned manually before. Tuning by indexes proposed by Squash yielded a reduction to 0:26 min.
- 5.c shows a Sequest™ query with partitions proposed by Squash. The execution time was 1:53 min (manually tuned), whereas tuning by Squash and application of partitions achieved further reduction of about 19% to 1:33 min.

Horizontal Partitioning

The core task for partitioning a table horizontally is to suggest a column that is suitable for partitioning.

- In the case of *range* partitioning, the borders of each partition are calculated by analysing the histograms of the partition key.
- For *hash* partitioning, the number of partitions is calculated from the table volume.

6 Conclusion

We have presented the extensible Squash framework, which supports relational database analysis, tuning, and refactoring. The input and ouput data structures of Squash are XML–based, and operations on these data are performed using the declarative languages PROLOG and FNQuery.

During the refactoring of database applications, Squash considers the interactions between application code and database schema definition; both can be handled and manipulated automatically. Thus, application maintenance is simplified, and the risk of making errors is reduced, in sum yielding time and resource savings. In a case study, Squash could improve some of the tuning proposals of a human database administrator, and yielded at least as efficient proposals for most examples.

In the future we are planning to analyze and refactor database applications where the SQL code can be *embedded* in other programming languages – such as Java and PHP – as well, based on XML representations of the abstract syntax trees, which we have already developed [28]. Then Squash will be able to derive the workload of the application being analyzed at run–time, and thus can predict dynamically generated SQL statements as well as the contents of bind variables. The application log will not become useless, because the frequency of the SQL commands must still be derived by analyzing an accounting protocol.

By analyzing the source code of the application, Squash will also be able to cope with the use of query cursors and other types of embedded SQL statements executed in other types of loops. Following this approach, dynamic SQL statements and queries involved in nested loops can be analyzed. This is especially important for updates or inserts depending on preceding select statements. Additionally, the use of nested cursors is a common stategy for tuning the run–time performance of database applications, which Squash will then be able to handle directly.

References

1. Agrawal, S., Narasayya, V., Yang, B.: Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In: Proc. ACM SIGMOD Intl. Conference on Management of Data, pp. 359–370. ACM Press, New York (2004)
2. Bellatreche, L., Karlapalem, K., Mohania, M.K., Schneider, M.: What Can Partitioning Do for Your Data Warehouses and Data Marts? In: Proc. Intl. Symposium on Database Engineering and Applications (IDEAS 2000), pp. 437–446. IEEE Computer Society Press, Los Alamitos (2000)
3. Boehm, A.M., Sickmann, A.: A Comprehensive Dictionary of Protein Accession Codes for Complete Protein Accession Identifier Alias Resolving. Proteomics 6(15), 4223–4226 (2006)
4. Brass, S., Goldberg, C.: Proving the Safety of SQL Queries. In: Proc. 5th Intl. Conference on Quality of Software (2005)
5. Chamberlin, D.: XQuery: a Query Language for XML. In: Proc. ACM SIGMOD Intl. Conference on Management of Data 2003, pp. 682–682. ACM Press, New York (2003)
6. Chaudhuri, S., Narasayya, V.: Autoadmin What–If Index Analysis Utility. In: Proc. Intl. Conference on Management of Data Archives. Proc. ACM SIGMOD Intl. Conference on Management of Data 1988, pp. 367–378. ACM Press, New York (1988)

7. Chaudhuri, S., Weikum, G.: Rethinking Database System Architecture: Towards a Self–Tuning RISC–Style Database System. In: Proc. 26th Intl. Conference on Very Large Data Bases (VLDB), pp. 1–10 (2000)
8. Choenni, S., Blanken, H.M., Chang, T.: Index Selection in Relational Databases. In: Proc. 5th Intl. Conference on Computing and Information (ICCI), pp. 491–496. IEEE, Los Alamitos (1993)
9. Clocksin, W.F., Mellish, C.S.: Programming in PROLOG, 5th edn. Springer, Heidelberg (2003)
10. Dias, K., Ramacher, M., Shaft, U., Venkataramani, V., Wood, G.: Automatic Performance Diagnosis and Tuning in Oracle. In: Proc. 2nd Biennal Conference on Innovative Data Systems Research (CIDR), pp. 84–94 (2005)
11. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 5th edn. Benjamin/Cummings (2006)
12. Eng, J.K., McCormack, A.L., Yates, J.R.: An Approach to Correlate Tandem Mass Spectral Data of Peptides with Amino Acid Sequences in a Protein Database. Journal of the American Society for Mass Spectrometry 5(11), 976–989 (1994)
13. Fagin, R.: Normal Forms and Relational Database Operators. In: Proc. ACM SIGMOD Intl. Conference on Management of Data (1979)
14. Finkelstein, S., Schkolnick, M., Tiberio, P.: Physical Database Design for Relational Databases. ACM Transactions on Database Systems (TODS) 13(1), 91–128 (1988)
15. Goldberg, C., Brass, S.: Semantic Errors in SQL Queries: A Quite Complete List. In: Proc. 16. GI–Workshop Grundlagen von Datenbanken, pp. 58–62 (2004)
16. Gruenwald, L., Eich, M.: Selecting a Database Partitioning Technique. Journal of Database Management 4(3), 27–39 (1993)
17. Intl. Organization for Standardization: ISO/IEC 9075–14:2003 Information Technology – Database Languages – SQL – Part 14: XML Related Specifications, SQL/XML (2003)
18. Kwan, E., Lightstone, S., Schiefer, B., Storm, A., Wu, L.: Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. In: 10. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW, Datenbanksysteme für Business, Technologie und Web), Bd. 26. LNI, pp. 620–629 (2003)
19. Perkins, D.N., Pappin, D.J.C., Creasy, D.M., Cottrell, J.S.: Probability–Based Protein Identification by Searching Sequence Databases Using Mass Spectrometry Data. Electrophoresis 20(18), 3551–3567 (1999)
20. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd edn. McGraw-Hill, New York (2003)
21. Rozen, S., Shasha, D.: A Framework for Automating Physical Database Design. In: Proc. 17th Intl. Conference on Very Large Data Bases (VLDB), pp. 401–411. Morgan Kaufmann, San Francisco (1991)
22. Seipel, D.: Processing XML Documents in PROLOG. In: Proc. 17th Workshop on Logic Programmierung (WLP) (2002)
23. Seipel, D., Baumeister, J., Hopfner, M.: Declarative Querying and Visualizing Knowledge Bases in XML. In: Proc. 15th Intl. Conference on Declarative Programming and Knowledge Management (INAP), pp. 140–151 (INAP)
24. Telford, R., Horman, R., Lightstone, S., Markov, N., O'Connell, S., Lohman, G.: Usability and Design Considerations for an Autonomic Relational Database Management System. IBM Systems Journal 42(4), 568–581 (2003)
25. Wielemaker, J.: An Overview of the SWI– PROLOG Programming Environment. In: Proc. 13th Intl. Workshop on Logic Programming Environments (WLPE), pp. 1–16 (2003)
26. Wielemaker, J.: SWI – PROLOG. Version (2007), `http://www.swi-prolog.org/`

27. Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G., Skelley, V.: DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In: Proc. 16th Intl. Conference on Data Engineering, pp. 101–110 (2000)
28. Wahler, V., Seipel, D., von Gudenberg, W.J., Fischer, G.: Clone Detection in Source Code by Frequent Itemset Techniques. In: 4th IEEE Intl. Workshop on Source Code Analysis and Manipulation (SCAM), pp. 128–135 (2004)
29. Weikum, G., Hasse, C., Mönkeberg, A., Zabback, P.: The Comfort Automatic Tuning Project. Information Systems 19(5), 381–432 (1994)
30. Weikum, G., Mönkeberg, A., Hasse, C., Zabback, P.: Self–Tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In: Proc. 28th Intl. Conference on Very Large Data Bases (VLDB), pp. 20–31 (2002)
31. Zahedi, R.P., Sickmann, A., Boehm, A.M., Winkler, C., Zufall, N., Schönfisch, B., Guiard, B., Pfanner, N., Meisinger, C.: Proteomic Analysis of the Yeast Mitochondrial Outer Membrane Reveals Accumulation of a Subclass of Preproteins. Molecular Biology of the Cell 17(3), 1436–1450 (2006)

Relational Models for Tabling Logic Programs in a Database

Pedro Costa[1], Ricardo Rocha[2], and Michel Ferreira[1]

[1] DCC-FC & LIACC
University of Porto, Portugal
c0370061@dcc.fc.up.pt, michel@dcc.fc.up.pt
[2] DCC-FC & CRACS
University of Porto, Portugal
ricroc@dcc.fc.up.pt

Abstract. Resolution strategies based on tabling are considered to be particularly effective in Logic Programming. Unfortunately, when faced with applications that compute large and/or many answers, memory exhaustion is a potential problem. In such cases, table deletion is the most common approach to recover space. In this work, we propose a different approach, storing tables into a relational database. Subsequent calls to stored tables import answers from the database, rather than performing a complete re-computation. To validate this approach, we have extended the YapTab tabling system, providing engine support for exporting and importing tables to and from the MySQL RDBMS. Three different relational models for data storage and two recordset retrieval strategies are compared.

1 Introduction

Tabling [1] is an implementation technique where intermediate answers for subgoals are stored and then reused whenever a repeated call appears. Resolution strategies based on tabling [2,3] proved themselves particularly effective in Logic Programming – the search space is reduced, looping is avoided and the termination properties of Prolog models based on SLD resolution are enhanced.

The performance of tabled evaluation largely depends on the implementation of the table itself – being called very often, fast lookup and insertion capabilities are mandatory. Applications can make millions of different calls, hence compactness is also required. Arguably, the most successful data structure for tabling is the *trie* [4]. Tries meet the previously enumerated criteria of compactness and efficiency quite well. The YapTab tabling system [5] uses tries to implement tables.

Used in applications that pose many queries, possibly with a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory. Swapping the main memory to disk may attenuate the problem but it is not a practical solution, since the memory space is always limited by the underlying architecture. In general, there is no choice but to throw away some of those tables, ideally, the least likely to be used next [6]. The most common control mechanism implemented in most tabling systems is to have a set of

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 99–116, 2009.

tabling primitives that the programmer can use to dynamically abolish some of the tables. A more recent proposal has been implemented in YapTab [6], where a memory management strategy based on a *least recently used* algorithm automatically recovers space among the least recently used tables when memory runs out. With this approach the programmer can still force the deletion of particular tables, but may also transfer the decision of which tables are potential candidates for deletion to the memory management algorithm. Note that, in both situations, the loss of answers stored within the deleted tables is unavoidable, eventually leading to re-computation.

In this work, we propose an alternative approach in which tables are sent to secondary memory – a relational database management system (RDBMS) – rather than deleted. Later, when a repeated call occurs, the answers are reloaded from the database thus avoiding re-computation. With this approach, the memory management algorithm can still be used, this time to decide which tables are to be sent to the database when memory runs out. To validate this approach we propose DBTab [7], a relational model for tabled logic program support resulting from the coupling of the YapTab tabling system with the MySQL RDBMS. In this initial implementation the ability of DBTab to represent terms is restricted to atomic strings and numbers.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we introduce our model and discuss how tables can be represented in a RDBMS. We then describe how we have extended YapTab to provide engine support for database stored answers. Finally, we present initial results and outline some conclusions.

2 The Table Space

Tabled programs evaluation proceeds by storing all computed answers for current subgoals in a proper data space, the *table space*. Whenever a subgoal $\mathcal{S}$ is called for the first time, a new table entry is allocated in the table space – all answers for subgoal $\mathcal{S}$ will be stored under this entry. Variant calls to $\mathcal{S}$ are resolved consuming those previously stored answers. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant subgoals. When all possible answers are found, $\mathcal{S}$ is said to be *completely evaluated.*

The table space may be accessed in a number of ways: **(i)** to find out if a subgoal is in the table and, if not, insert it; **(ii)** to verify whether a newly found answer is already in the table and, if not, insert it; and **(iii)** to load answers to variant subgoals. With these requirements, performance becomes an important issue so YapTab implements its table space resorting to *tries* [8] – which is regarded as a very efficient way to implement tables [4].

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term. At the entry point we have the root node. Internal nodes store tokens in terms and leaf nodes specify the end of terms. Each root-to-leaf path represents a term described by the tokens labelling the nodes traversed. For example, the tokenized form of the term $p(X, q(Y, X), Z)$

is the stream of 6 tokens: $p/3, X, q/2, Y, X, Z$. Two terms with common prefixes will branch off from each other at the first distinguishing token.

The internal nodes of a trie are 4-field data structures. One field stores the node's token, one second field stores a pointer to the node's first child, a third field stores a pointer to the node's parent and a fourth field stores a pointer to the node's next sibling. Each internal node's outgoing transitions may be determined by following the child pointer to the first child node and, from there, continuing through the list of sibling pointers.

To increase performance, YapTab enforces the *substitution factoring* [4] mechanism and implements tables using two levels of tries - one for subgoal calls, the other for computed answers. More specifically, the table space of YapTab is organized in the following way:

- each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate's *subgoal trie*;
- each different subgoal call is represented as a unique path in the subgoal trie, starting at the predicate's table entry and ending in a *subgoal frame* data structure, with the subgoal sub-terms being stored within the path's internal nodes;
- the *subgoal frame* data structure acts as an entry point to the *answer trie*;
- each different subgoal answer is represented as a unique path in the *answer trie*, starting at a particular leaf node and ending at the subgoal frame. Oppositely to subgoal tries, answer trie paths hold just the substitution terms for unbound variables in the corresponding subgoal call.
- the leaf's child pointer of answers are used to point to the next available answer, a feature that enables answer recovery in insertion order. The subgoal frame has internal pointers that point respectively to the first and last answer on the trie. Whenever a variant subgoal starts consuming answers, it sets a pointer to the first leaf node. To consume the remaining answers, it must follow the leaf's linked chain, setting the pointer as it consumes answers along the way. Answers are loaded by traversing the answer trie nodes bottom-up.

An important point when using tries to represent terms is the treatment of variables. We follow the formalism proposed by Bachmair *et al.* [9], where each variable in a term is represented as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $VAR0, ..., VARN$, such that $numbervar(X) < numbervar(Y)$ if X is encountered before Y in the left-to-right traversal of t.

An example for a tabled predicate $f/2$ is shown in Fig. 1. Initially, the subgoal trie contains only the root node. When the subgoal $f(X, a)$ is called, two internal nodes are inserted: one for the variable X, and a second for the constant a. The subgoal frame is inserted as a leaf, waiting for the answers. Then, the subgoal $f(Y, 1)$ is inserted. It shares one common node with $f(X, a)$ but, having a different second argument, a new subgoal frame needs to be created. Next, the answers for $f(Y, 1)$ are stored in the answer trie as their values are computed.

Two facts are noteworthy in the example of Fig. 1. First, the example helps us to illustrate how the state of a subgoal changes throughout execution. A subgoal is in the *ready* state when its subgoal frame is added. The state changes to *evaluating* while new answers are being added to the subgoal's table. When all possible answers are stored in the table, the state is set to *complete*. When not complete or evaluating, the subgoal is said to be *incomplete*.

Second, some terms require special handling from YapTab, such as the 2^{30} integer term, appearing in the picture surrounded by the FLI (*functor long integer*) markers. This is due to the node's design. Internally, YapTab terms are 32 (or 64) bit words, divided in two areas: type and value. Hence, the value part, which is used for data storage purposes, is always smaller than an equally typed C variable. Henceforth, we shall refer to terms whose value fits the proper slot, such as small integer and atom terms, as *short atomic terms* and to all the others, such as floating-point and large integers, as *long atomic terms*. Given a long atomic term, YapTab **(i)** ignores the term's type and forces its (full) value into a single node, or **(ii)** splits the value into as many pieces as needed, assigning a new node to each and every one of these. Either way, the type bits are used to store value data so additional nodes, holding special type markers, are used to delimit the term's value.

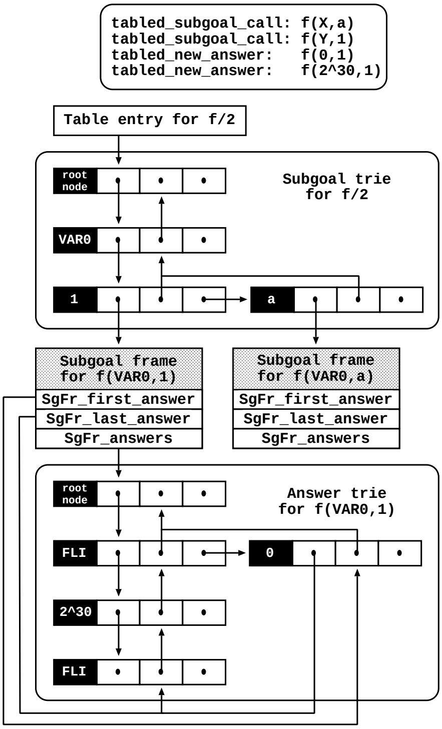

tabled_subgoal_call: f(X,a)
tabled_subgoal_call: f(Y,1)
tabled_new_answer: f(0,1)
tabled_new_answer: f(2^30,1)
Table entry for f/2
root node
Subgoal trie for f/2
VAR0
1
a
Subgoal frame for f(VAR0,1)
SgFr_first_answer
SgFr_last_answer
SgFr_answers
Subgoal frame for f(VAR0,a)
SgFr_first_answer
SgFr_last_answer
SgFr_answers
root node
Answer trie for f(VAR0,1)
FLI
0
2^30
FLI

Fig. 1. The table space organization

3 The DBTab Relational Model

The choice of effective models for table space representation is a hard task to fulfill. The relational model is expected to allow quick storage and retrieval operations over tabled answers in order to minimize the impact on YapTab's performance. Conceptually, DBTab's relational model is straightforward – every tabled subgoal is mapped into a database relation. The relation's name bears the predicate's functor and arity and its attributes match the predicate's arguments. Two distinct database models spawned from this basic concept.

3.1 Datalog Model - Tries as Sets of Subgoals

Given an in-memory tabled subgoal, the complete set of its known answers may be regarded as a set of ground facts that are known to hold in a particular execution context. In other words, known subgoal answers are regarded as Datalog facts. Thus, and henceforth, this schema is referred to as the *Datalog model*.

In this model, each tabled predicate p/n is mapped into a relational table SESSIONk_PN, where k is the current session identifier. Predicate arguments become the ARGi integer fields and the INDEX field is used to store the answer's index in the answer chain of the in-memory subgoal trie. One can further refine this model by choosing one of the two following alternatives.

Inline Values. As previously stated, DBTab maps tabled predicates directly into database relations bearing the same name and argument number. Thus, predicates appear as tables, subgoals appear as records within those tables and subgoal sub-terms appear as constants in record fields. Unfortunately, the different nature and size of logical atoms presents a problem. Table fields may be given a primitive type big enough to accommodate all kinds of atoms, but that seems to be too expensive and unpractical.

Instead, each of the predicate's terms is mapped into a set of fields, one for each supported atomic type. Whenever storing a subgoal answer, the term's type is checked to decide which table field must be used to hold the term's value; the remaining ones are set to NULL. This policy also simplifies the reverse decision process – since all of these fields are regarded as a single predicate's argument, the value for the stored term is always the non-NULL one. Due to their size, short atomic terms may be directly stored within the corresponding ARGi integer fields. This small optimization enables the shortening of each field set by one. Figure 2 displays an example of such a relation for the $f/2$ tabled predicate, introduced back in Fig. 1.

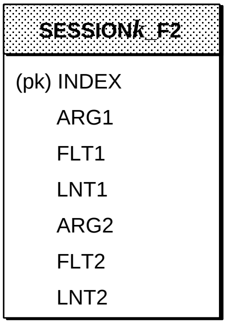


Fig. 2. The inline values model

Separate Values. The inline values model produces highly sparse tables, since a lot of storage space is wasted in records containing mostly NULL-valued fields. To take full advantage of the normalized relational model, the subgoal's answer terms may be stored in several auxiliary tables according to their logical nature. Figure 3 shows the separate values relational model for the previously stated $f/2$ tabled predicate.

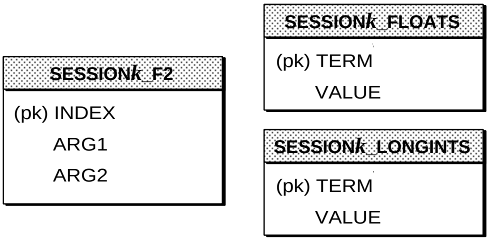


Fig. 3. The separate values model

Again, short atomic terms are directly stored within the ARGi integer fields, thus reducing the number of required auxiliary tables. Long atomic terms, on the other hand, are stored in auxiliary tables, such as SESSIONk_LONGINTS and SESSIONk_FLOATS. Each long atomic value appears only once and is identified by a unique key value that replaces the term's value in the proper field ARGi of SESSIONk_F2. The key is in fact a YapTab term, whose type bits tell which auxiliary table holds the actual primitive term's value whereas the value bits hold a sequential integer identifier. Despite the practical similarity, these keys may not be formally defined as foreign keys – notice that the same ARGi field may hold references to both auxiliary tables.

3.2 Hierarchical Model - Tries as Sets of Nodes

Despite its correct relational design, the Datalog model drifts away from the original concept of tries. From the inception, tabling tries where though of as compact data structures. This line of reasoning can also be applied to any proposed data model.

Storing the complete answer set extensionally is a space (and time) consuming task, hence one should draw inspiration from the substitution factoring of tabling tries [4], keeping only answer tries in the database. This not only significantly reduces the amount of data in transit between the logical and the database engines, but also perfectly suits YapTab's memory management algorithm [6], which cleans answer tries from memory but maintains subgoal tries intact.

The challenge is then to figure out a way to mimic the hierarchical structure of a trie in the flat relational database environment. Tries may be regarded as partially ordered sets of nodes ordered by reachability. Hence, mapping relations must be defined in such a way that its tuples contain enough information to preserve the nodes' hierarchy. Such a model is henceforth referred to as the *hierarchical model*.

Arguably, the most simple and compact way to represent a trie is through a database relation like the one presented in Fig. 4, where every node knows only its own order, its parent's order, and its own token. Of all possible numbering methods, we feel that assigning nodes' order while traversing the trie in post-order is the most appropriate, since it will eventually lead to a reduction of the number of data transactions, while maintaining the in-memory insertion order at the same time.

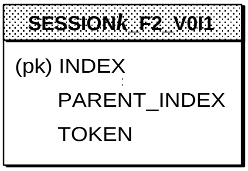


Fig. 4. The hierarchical model

In this context, mapping relations no longer hold entire predicate answer sets, rather each relation stands for a single subgoal call. Hence, one must be able to tell the different subgoals apart. A straightforward way to establish this distinction is to encapsulate the arguments' values as a suffix to the relation's name. The chosen naming convention dictates that each argument is identified by a capital letter, indicating the type of the term, and a number, indicating its order within the subgoal. The possible letters are I and L for standard and long integers, D for doubles, A for (string) atoms and V for (unbound) variables. Figure 4 displays an example of such a relation for the $f/2$ tabled predicate.

4 Extending YapTab's Design

From the beginning, our goal was to introduce DBTab without disrupting the previously existing YapTab framework. With that in mind, we have striven to keep both tabling semantics and top-level predicates syntax intact. Rather, top-level predicates and internal instructions were internally modified to include calls to the developed API functions.

The dumping of in-memory answer tries into the database is triggered by the *least recently used* algorithm when these tables are selected for destruction. Recordset reloading takes place when calls to tabled subgoals occur and their tables have been previously dumped.

Communication between the YapTab engine and the RDBMS is mostly done through the MySQL C API for prepared statements. Two of the major table space data structures, *table entries* and *subgoal frames*, are expanded with new pointers (see Fig. 5) to proprietary `PreparedStatement` wrapping data structures. These structures hold specialized queries that enable the dumping and loading of subgoals to and from the database. All SQL statements are executed in a transactional context.

Two new predicates are added to manage database session control. To start a session one must call the `tabling_init_session/2` predicate. It takes two arguments, the first being a database connection handler and the second being a session identifier. The connection handler is usually created resorting to the Myddas package [10] and the session identifier can be either a unbound variable or a positive integer term, meaning respectively that a new session is to be initiated or a previously created one is to be reestablished. The `tabling_kill_session/0` terminates the currently open session.

YapTab's directive `table/1` is used to set up logic predicates for tabling. The choice of a storage model determines the behaviour of DBTab's expanded version of this built-in predicate. In the hierarchical model, it issues a table

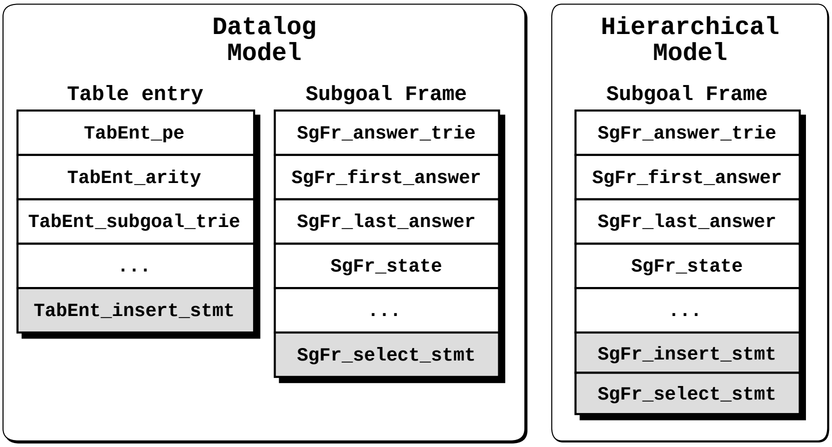


Fig. 5. New table frames' fields

creation statement and exists. In the Datalog model (both alternatives), an additional step creates an INSERT prepared statement. The statement, placed inside the predicate's table entry in the new `TabEnt_insert_stmt` field, allows the introduction of any subgoal answer for the tabled predicate. Two examples of such statements for the inline and separate values models, labeled **(1)** and **(2)** respectively, are shown in Fig. 6.

```
(1) INSERT IGNORE
    INTO SESSIONk_F2(INDEX,ARG1,FLT1,LNT1,ARG2,FLT2,LNT2)
    VALUES (?,?,?,?,?,?,?);
(2) INSERT IGNORE
    INTO SESSIONk_F2(INDEX,ARG1,ARG2)
    VALUES (?,?,?);
(3) INSERT IGNORE
    INTO SESSIONk_F2_V0I1 (INDEX, PARENT_INDEX, TOKEN)
    VALUES (?,?,?);
(4) SELECT DISTINCT ARG1,LNT1
    FROM SESSIONk_F2
    WHERE ARG2=14;
(5) SELECT DISTINCT F2.ARG1 AS ARG1, L.VALUE AS LNT1
    FROM SESSIONk_F2 AS F2
            LEFT JOIN SESSIONk_LONGINTS AS L ON (F2.ARG1=L.TERM)
    WHERE F2.ARG2=14
    ORDER BY F2.INDEX;
(6) SELECT INDEX, TOKEN
    FROM SESSIONk_F2_V0I1
    WHERE PARENT_INDEX=?;
```

Fig. 6. Prepared statements for $f(Y, 1)$

The `abolish_table/1` built-in predicate is used to destroy in-memory tables for tabled predicates. The DBTab expanded version of this predicate frees all prepared statements used to manipulate the associated database tables and drops them if they are no longer being used by any other instance.

Besides the referred predicates, the tabling instruction set must suffer some small modifications. The instruction responsible for the subgoal insertion into the table space is adapted to create a SELECT prepared statement. This statement, kept inside the subgoal frame structure in the new `SgFr_select_stmt` field, is used to load answers for that subgoal. Examples of such statements for the inline values, separate values and hierarchical models, labeled **(4)**, **(5)** and **(6)** respectively, are shown in Fig. 6. The selection query is tailored to reduce the search space – this is accomplished by including all bound terms in comparison expressions within the WHERE sub-clause[1]. Notice that SELECT statement of the Datalog models, examples **(4)** and **(5)**, bear a DISTINCT option. This modifier is added to enforce the uniqueness of retrieved answers.

[1] Here, we are considering a 32 bit word representation where the integer term of value 1 is internally represented as 14.

In the hierarchical model, an additional step creates the INSERT prepared statement. This statement is placed inside the subgoal `SgFr_insert_stmt` new field. Unlike the Datalog models, the hierarchical model's table name identifies unequivocally the subgoal for which its stands, rendering the statement useless to all other subgoals belonging to the same predicate. Example **(3)** of Fig. 6 illustrates the hierarchical INSERT statement.

4.1 Exporting Answers

When the system runs out of memory, the *least recently used* algorithm is called to purge some of tables from the table space. The algorithm divides subgoal frames in two categories, *active* and *inactive*, according to their execution state. Subgoals in *ready* and *incomplete* states are considered *inactive*, while subgoals with *evaluating* state are considered *active*. Subgoals in the *complete* state may be either active or inactive.

Figure 7 illustrates the memory recovery process. Subgoal frames corresponding to inactive subgoals are chained in a double linked list. Two global registers point to the most and least recently used inactive subgoal frames. Starting from the least recently used, the algorithm navigates through the linked subgoal frames until it finds a memory page that fits for recovery. Only tables storing more than one answer may be elected for space recovery (completed nodes with a yes/no answer are ignored). Recall that only the answer trie space is recovered. Rocha [6] suggests that for a large number of applications, these structures consume more that 99% of the table space.

DBTab cuts in before the actual table elimination. At that point, a specific API function begins a new data transaction. From this point on, implementations differs according to the used database model.

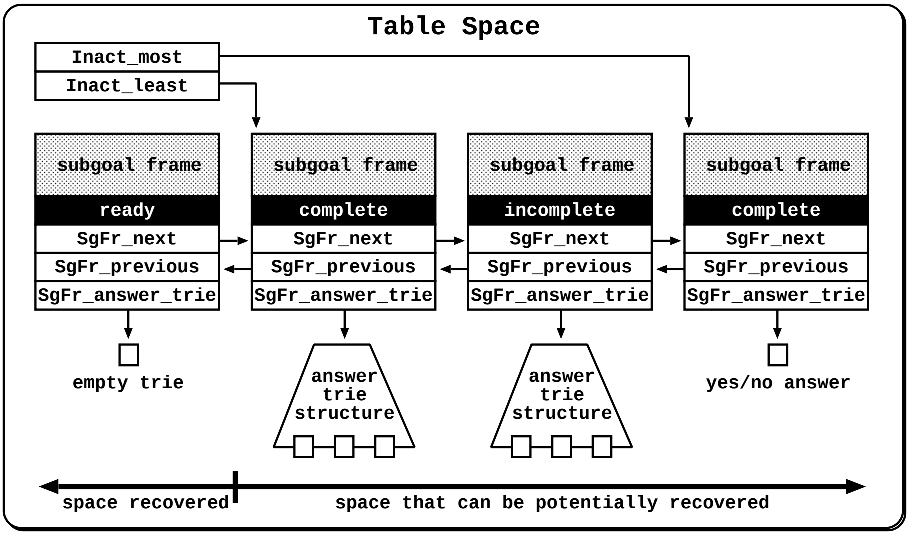


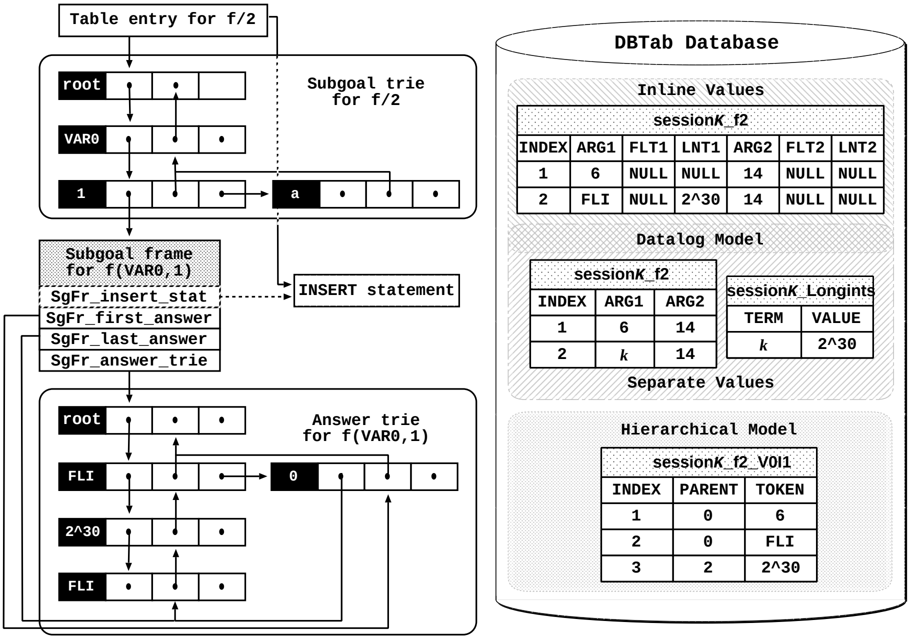
Fig. 7. The least recently used algorithm in action

The hierarchical model implementation traverses the trie in post-order, numbering and storing nodes as it goes along. The complete set of records belonging to an answer trie is stored recursively. Starting from a level's last node (first record), the next trie level is stored before proceeding to the previous node (next record). The table records are created as the trie nodes are visited.

The Datalog model implementation first traverses the subgoal trie branch bottom-up (starting from the subgoal frame), binding every ground term it finds along the way to the respective parameter in the INSERT statement. When the root node is reached, all parameters consisting of variable terms will be left NULL. The attention is then turned to the answer trie and control proceeds cycling through the terms stored within the answer trie nodes. Again, the remaining NULL statement parameters are bound to the current answer terms and the prepared statement is executed, until no more answers remain to be stored. The last step consists in constructing the specific SELECT statement to be used to fetch the answers for the subgoal, whose ground terms are used to refine the search condition within the WHERE sub-clause.

With both models, the transaction is committed and the subgoal frame changes its internal state to *stored*, signalling that its known answers now reside on a database table. Finally, the least recently used algorithm resumes its normal behaviour by removing the answer trie from memory.

Figure 8 illustrates the final result of the described process using both storage models. The branch for the integer term of value 0 is stored first, and the



sessionK_f2						
INDEX	ARG1	FLT1	LNT1	ARG2	FLT2	LNT2
1	6	NULL	NULL	14	NULL	NULL
2	FLI	NULL	2^30	14	NULL	NULL

sessionK_f2		
INDEX	ARG1	ARG2
1	6	14
2	*k*	14

sessionK_Longints	
TERM	VALUE
k	2^30

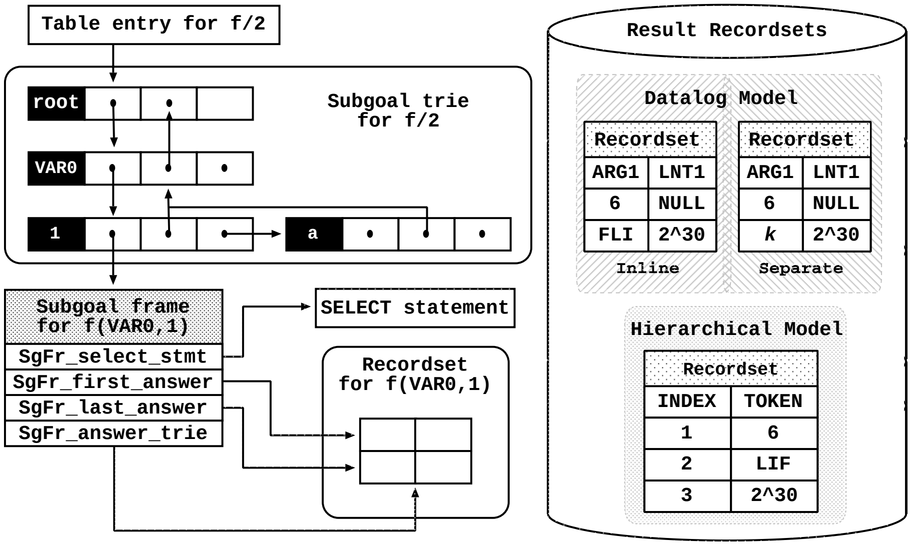
sessionK_f2_V0I1		
INDEX	PARENT	TOKEN
1	0	6
2	0	FLI
3	2	2^30

Fig. 8. Exporting $f(Y, 1)$ using both storage models

branch for the long integer term 2^{30} is stored next[2]. Notice how, in the separate values model, the ARG1 field of the second record holds the key for the auxiliary table record. Also, notice how the hierarchical model saves one record omitting the second FLI type marker. Recall that, in this model, the INSERT prepared statement is placed at the subgoal frame level.

4.2 Importing Answers

After memory recovery, a repeated call to a pruned subgoal may occur. Normally, YapTab checks the subgoal frame state and, should it be ready, recomputes the entire answer trie before proceeding. Again, DBTab cuts in before this last step. An API function uses the SELECT prepared statement residing in the subgoal frame to retrieve the previously computed answers from the database and uses them to rebuild the answer trie. Figure 9 shows, in the right boxes, the resulting *views* for each storage model. The way in which the returned recordset is interpreted differs from model to model.



Recordset	
ARG1	LNT1
6	NULL
FLI	2^30

Recordset	
ARG1	LNT1
6	NULL
k	2^30

Recordset	
INDEX	TOKEN
1	6
2	LIF
3	2^30

Fig. 9. Importing $f(Y, 1)$ using both storage models

The hierarchical model reloading operation is straightforward. The recordset is loaded one level at a time from the database. All records within a certain level are fetched in insertion order simply by ordering the selection by the index field. The complete set of records belonging to an answer trie is fetched recursively. Starting from a level's last node (first record), the next trie level is fetched before proceeding to the previous node (next record). The trie nodes are created as the records are consumed. A special feature in this process involves the setting and

[2] Again, we are considering a 32 bit word representation where the integer terms of value 0 and 1 are internally represented respectively as 6 and 14.

reading of a type flag whenever an opening long atomic term delimiter is found. The flag is set to the proper primitive type so that, once the entire term is reloaded from the database, the closing delimiter is inserted into the trie. This allows a small efficiency gain, since it reduces tables' and views' sizes.

The Datalog models focus on the ARGi fields, where no NULL values can be found. Additional columns, placed immediately to the right of the ARGi fields, are regarded as possible placeholders of answer terms only when the first field conveys long atomic type markers. In such a case, the non-NULL additional field value is used to create the specific YapTab term. The separate values alternative uses the auxiliary table's key value as marker, while the inline values alternative uses the special type markers directly.

Complete answer-sets present themselves as a special case of the import operation. Since they will no longer change, one wonders if trie reconstruction is the best approach for most of the cases. In both Datalog alternatives, two possible strategies may be used to supply the YapTab engine with the answers fetched by the SELECT statement.

Rebuilding the Answer Trie. The retrieved recordset is used to rebuild the answer trie and is then discarded. Records are sequentially traversed and each one of them provides the sub-terms of a complex term – the answer. This term is passed to the ancillary table look-up/insertion function that places the new answer in the respective subgoal table. By the end of the process, the entire answer trie resides in the table space and the recordset can then be released from memory. This approach requires small changes to YapTab and mostly makes use of its already implemented API.

Browsing the Record-Set. In this strategy, the retrieved recordset is kept in memory. Since the answer tries will not change once completed, all subsequent subgoal calls may fetch their answers browsing the recordset through offset arithmetics. Figure 9 illustrates how the ancillary YapTab constructs are used to implement this idea. The left side box presents the state of the subgoal frame after answer collection for $f(Y, 1)$. The internal pointers are set to the first and last rows of the recordset. When consuming answers, the first record's offset along with the subgoal frame address are stored in a *loader choice point*[3]. The fetched record and its field values are then used to bind the free variables found for the subgoal in hand. If backtracking occurs, the choice point is reloaded and the last recorded offset is used to step through to the next answer. When, at the end of the recordset, an invalid offset is reached, the loader choice point is discarded and execution fails, thus terminating the on-going evaluation. It is expected that this approach may introduce a small gain in performance and memory usage because **(i)** retrieval transaction occurs only once; **(ii)** no time and memory are spent rebuilding the answer trie; and **(iii)** long atomic term representation required down to one fourth of the usually occupied memory.

[3] A loader choice point is a WAM choice point augmented with a pointer to the subgoal frame data structure and with the offset for the last consumed record.

```
% connection handle creation stuff
:- consult(’graph.pl’).
:- tabling_init_session(Conn,Sid).

% top query goal
go(N) :- statistics(walltime, [Start,_]), benchmark(N),
         statistics(walltime, [End,_]), Time is End-Start,
         writeln([’WallTime is ’,Time]).
benchmark(1):- path(A,Z), fail.
benchmark(1).
benchmark(2):- write_path(A,Z), fail.
benchmark(2).

% path(A,Z) and write_path(A,Z) succeed if there is a path between A and Z
:- table path/2.
path(A,Z):-path(A,Y), edge(Y,Z).
path(A,Z):-edge(A,Z).

:- table write_path/2.
write_path(A,Z):- write_path(A,Y), edge(Y,Z), writeln([’(’,A,’,’,Z,’)’]).
write_path(A,Z):- edge(X,Y), writeln([’(’,A,’,’,Z,’)’]).
```

Fig. 10. Test program

5 Initial Experimental Results

Three main series of tests were performed in YapTab, both with and without the DBTab extensions. This meant to establish the grounds for a correct comparison of performances. The environment for our experiments was an Intel Pentium®4 2.6GHz processor with 2 GBytes of main memory and running the Linux kernel-2.6.18. DBTab was deployed on a MySQL 5.0 RDBMS running an InnoDB engine. Both engines were running on the same machine.

A path discovery program, presented in Fig. 10, was used to measure both YapTab and DBTab performances in terms of answer time. The program's main goal is to determine all existing paths for the graph presented in Fig. 11, starting from any node in the graph.

The *go*/1 predicate is the top query goal. It determines the benchmark predicates' performance by calculating the difference between the program's up-time before and after the benchmark execution. Benchmark predicates are *failure driven loops*, i.e., predicates defined by two clauses – a first one executes a call to the tabled

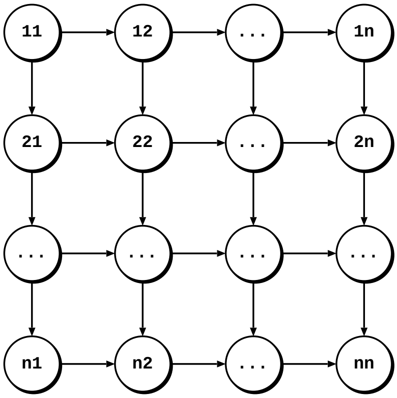


Fig. 11. Test graph

goal followed by a failure instruction and a second one ensures the successful completion of the top query goal.

Two tabled predicates were used to determine all existing paths in the loaded graph. Predicate *path*/2, henceforth called *benchmark #1*, helped us to establish minimum performance standards, since its answer-set is relatively easy both to obtain and handle through the use of standard tabling. However, every day problems often require heavy calculations or some sort of input/output operation. With this in mind, predicate *write_path*/2, henceforth called *benchmark #2*, was introduced. It basically results from the addition of a call to an output predicate, *writeln*/1, at the very end of both clauses of *path*/2. Our aim was to determine if DBTab could improve the overall performance of the main program when such heavy operations were performed. Note that from the start, it was our firm conviction that, for programs mostly based on *pure* tabled computations, DBTab would not bring any gain than improvement in program termination due to the overhead induced by the YapTab/MySQL communication.

For a relatively accurate measure of execution times, the node's type changed twice and the graph's size changed twenty-five times. Both benchmark programs were executed twenty-five times for each combination of type and size. The average of every combination's run-time was measured in milliseconds.

Early tests uncovered a major bottleneck during answer storage. Sending an answer a time to the database significantly damaged performance. As a result, DBTab now incorporates a MySQL feature that enables clustered insert statements, sending answer-tries to the database in clusters of 25 tuples.

Figure 12 shows three charts that visually summarize the results. The first chart shows that input/output operations introduce a significant decrease in performance. When comparing both benchmark's performance in YapTab, one observes that in average *benchmark #2* executes 87 times slower than the *benchmark #1*. The chart shows that the gap widens as the number of nodes in the answer trie grows.

The second chart in Fig. 12 shows first call execution times for *benchmark #2*. In order to measure the impact of DBTab on YapTab's performance, a full table dump is forced after each answer-set is complete. As expected, some overhead is induced into YapTab's normal performance. For the inline values model, the median storage overhead is 0.3 times the amount of time required to execute *benchmark #2*, within an interval ranging from 0.2 to 0.6 times. For the separate values alternative, the median overhead grows to 0.6 times, within an interval ranging from 0.3 to 0.7 times. For the hierarchical model, the median induced overhead is of 0.5 times the computation time of *benchmark #2*, within an interval ranging from 0.2 to 0.7 times. Additionally, label types also have impact on the execution times. In average, answer set generation using long integer labels is 1.6 times slower than when integer labels are used. No doubt, this is due to the larger number of nodes in the answer trie, which result in a larger number of transacted answers.

The third chart in Fig. 12 shows subsequent call execution times. From its observation, one can learn that retrieval operations have little cost in terms of

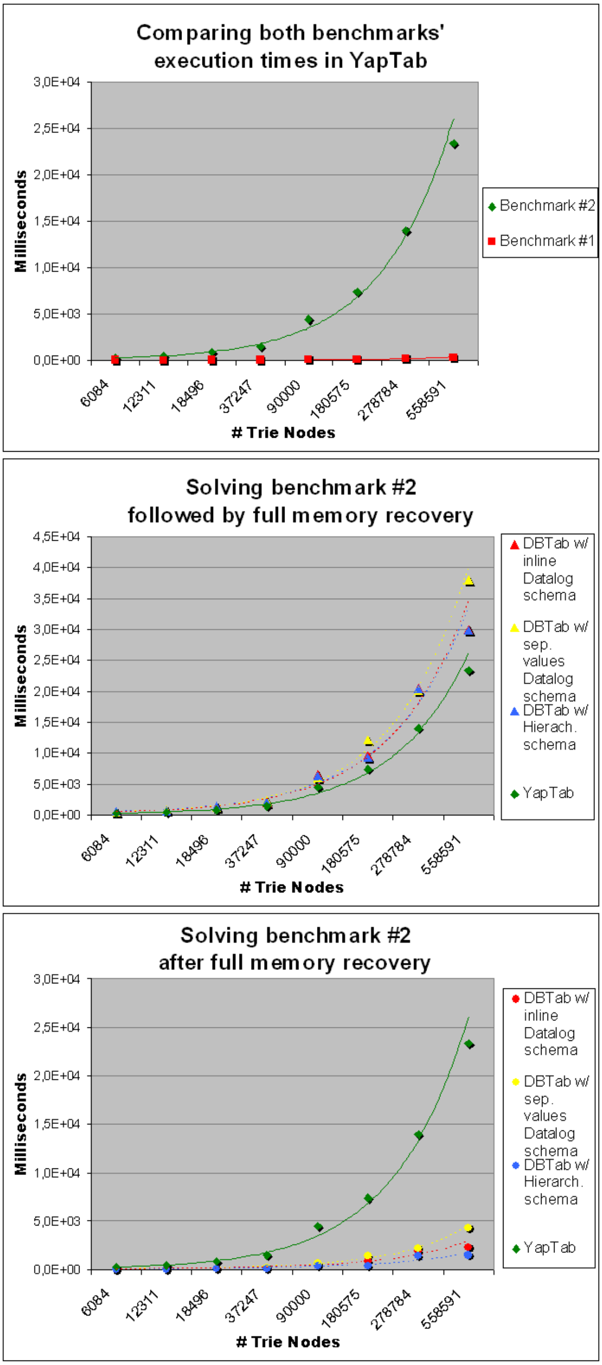


Fig. 12. Test results

performance and indeed introduce some speed-up when compared to full answer-set re-computation. For the inline values model, the minimum, median and maximum retrieval times are 0.1 times the amount of time required to execute *benchmark #2*. The performance of the other alternative is similar, although in the worst-case scenario the median retrieval time may rise to 0.2. For the hierarchical model, the median retrieval time is 0.1 times the computation time of *benchmark #2*, within an interval ranging from (nearly) 0.0 to 0.2 times.

The performances of trie traversal and recordset browsing are compared in Fig. 13. In this figure, it is possible to observe that recordset browsing times grow with the graph size. For the inline values model, it takes in average 3.8 times more the time required to traverse the respective answer trie. For the separate values, it takes only 3.6 times. However, this technique introduces a considerable gain in terms of used memory, as the recordset in-memory size is in average 0.4 of the correspondent answer trie size for both Datalog alternatives.

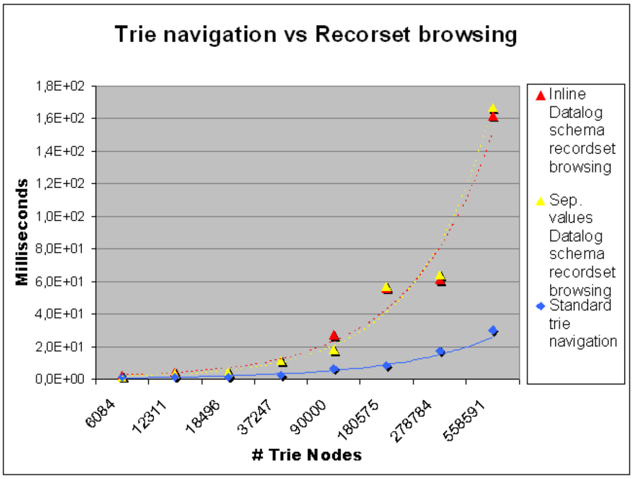


Fig. 13. Trie navigation versus recordset browsing

6 Conclusions and Further Work

In this work, we have introduced the DBTab model. DBTab was designed to be used as an alternative approach to the problem of recovering space when the tabling system runs out of memory. By storing tables externally instead of deleting them, DBTab avoids standard tabled re-computation when subsequent calls to those tables appear.

In all performed tests, data transaction performances revealed a similar pattern. Storage revealed to be an expensive operation. In all cases, this operation's cost exceeded that of recomputing the same answer set. The inline values model was always the fastest, the hierarchical model the second fastest and the separate values Datalog model was always the slowest. However, when the computation involved side-effected operations, this scenario has radically changed and the cost of storing the answer set became quite acceptable.

Things were somewhat different with the retrieval operation. Of all implementations, the hierarchical model was always the fastest and the separate values Datalog model the slowest. This last implementation shows a significant performance decay, no doubt induced by the use of left join clauses in the retrieval select statement. In average, the separate values Datalog model took at least twice the time to retrieve the answer-sets than its inline counterpart. When the answer set evaluation involved no costly operations, reloading answers from the database was obviously slower. On the other hand, when the side-effected operations were introduced, reloading became a quite attractive possibility.

In what regards to term types, integer terms where obviously the simplest and fastest to handle. The other primitive types, requiring special handling, induced significant overheads to both storage and retrieval operations. Atom terms, not considered in the tests, are known to behave as standard integers.

For small answer-sets, recordset browsing might be enough to locate a small initial subset of answers and decide whether that subset is the adequate one or if it should be ignored; in either case, this allows saving both the time and memory resources required for the complete trie reconstruction. However, as answer-sets size increase, this traversal approach performance decays, reaching up to three times the amount of time required to reconstruct the table. In this last case, the only advantage of recordset browsing is the introduced saving in terms of memory requirements.

Our preliminary results show that DBTab may become an interesting approach when the cost of recalculating a table largely exceeds the amount of time required to fetch the entire answer recordset from the database. As further work we plan to investigate the impact of applying DBTab to a more representative set of programs. We also plan to cover all possibilities for tabling presented by YapTab and extend DBTab to support lists and application terms.

Acknowledgments

This work has been partially supported by projects STAMPA (PTDC/EIA/ 67738/2006), JEDI (PTDC/EIA/66924/ 2006) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. Michie, D.: Memo Functions and Machine Learning. Nature 218, 19–22 (1968)
2. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: Shapiro, E. (ed.) ICLP 1986. LNCS, vol. 225, pp. 84–98. Springer, Heidelberg (1986)
3. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43, 20–74 (1996)

4. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming 38, 31–54 (1999)
5. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction, pp. 77–87 (2000)
6. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 155–169. Springer, Heidelberg (2006)
7. Costa, P., Rocha, R., Ferreira, M.: DBTAB: a Relational Storage Model for the YapTab Tabling System. In: Colloquium on Implementation of Constraint and LOgic Programming Systems, pp. 95–109 (2006)
8. Fredkin, E.: Trie Memory. Communications of the ACM 3, 490–499 (1962)
9. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993, FASE 1993, and TAPSOFT 1993. LNCS, vol. 668, pp. 61–74. Springer, Heidelberg (1993)
10. Soares, T., Ferreira, M., Rocha, R.: The MYDDAS Programmer's Manual. Technical Report DCC-2005-10, Department of Computer Science, University of Porto (2005)

Integrating XQuery and Logic Programming*

Jesús M. Almendros-Jiménez, Antonio Becerra-Terón,
and Francisco J. Enciso-Baños

Dpto. Lenguajes y Computación
Universidad de Almería
{jalmen,abecerra,fjenciso}@ual.es

Abstract. In this paper we investigate how to integrate the XQuery language and logic programming. With this aim, we represent XML documents by means of a logic program. This logic program represents the document schema by means of rules and the document itself by means of facts. Now, XQuery expressions can be integrated into logic programming by considering a translation (i.e. encoding) of *for-let-where-return* expressions by means of logic rules and a goal.

1 Introduction

The *eXtensible Markup Language (XML)* is a simple, very flexible text format derived from SGML. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. In this context, *XQuery* [W3C07b, CDF$^+$04, Wad02, Cha02] is a typed functional language devoted to express queries against XML documents. It contains *XPath* [W3C07a] as a sublanguage which supports navigation, selection and extraction of fragments from XML documents. *XQuery* also includes expressions (i.e. *for-let-where-return* expressions) to construct new XML values and to join multiple documents. The design of *XQuery* has been influenced by group members with expertise in the design and implementation of other high-level languages. *XQuery* has static typed semantics and a formal semantics which is part of the *W3C* standard [CDF$^+$04, W3C07b].

The integration of *logic programming languages* and *web technologies*, in particular, XML data processing is interesting from the point of view of the applicability of logic programming. On one hand, XML documents are the standard format of exchanging information between applications. Therefore, logic languages should be able to handle and query such documents. On the other hand, logic languages could be used for extracting and inferring semantic information from XML, RDF (*Resource Description Framework*) and OWL (*Ontology Web Language*) documents, in the line of *"Semantic Web"* requirements [BHL01]. Therefore, logic languages can find a natural and interesting application field in this area. The integration of *declarative programming* and *XML data processing*

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 117–135, 2009.

is a research field of increasing interest in the last years (see [BBFS05] for a survey). There are proposals of new languages for XML data processing based on functional, and logic programming.

The most relevant contribution is the *Galax* project [MS03, CDF$^+$04], which is an implementation of *XQuery* in functional programming, using *OCAML* as host language. There are also proposals for new languages based on functional programming rather than implementing *XPath* and *XQuery.* This is the case of *XDuce* [HP03] and *CDuce* [BCF05, BCM05], which are languages for XML data processing, using regular expression pattern matching over XML trees, subtyping as basic mechanism, and *OCAML* as host language. The *CDuce* language does fully statically-typed transformation of XML documents, thus guaranteeing correctness. In addition, there are proposals around *Haskell* for the handling of XML documents, such as *HaXML* [Thi02, ACJ04] and [WR99].

In the field of logic programming there are also contributions for the handling of XML documents. For instance, the *Xcerpt* project [SB02, BS02a] proposes a pattern and rule-based query language for XML documents, using the so-called query terms including logic variables for the retrieval of XML elements. For this new language, a specialized unification algorithm for query terms has been studied in [BS02b]. Another contribution of a new language is *XPathLog* (integrated in the the *Lopix* system) [May04] which is a *Datalog*-style extension for *XPath* with variable bindings. *Elog* [BFG01] is also a logic-based XML data manipulation language, which has been used for representing Web documents by means of logic programming. This is also the case of *XCentric* [CF07, CF03, CF04], which can represent XML documents by means of logic programming, and handles XML documents by considering terms with functions of flexible arity and regular types. *FNPath* [Sei02] is also a proposal for using *Prolog* as a query language for XML documents. It maps XML documents to a Prolog Document Object Model (DOM), which can consist of facts (graph notation) or a term structure (field notation). *FnPath* can evaluate XPath expressions based on that DOM. The *Rule Markup Language* (*RuleML*) [Bol01, Bol00b, Bol00a] is a different kind of proposal in this research area. The aim of *RuleML* is the representation of *Prolog* facts and rules in XML documents, and thus, the introduction of *rule systems* into the *Web.* Finally, some well-known *Prolog* implementations include libraries for loading and querying XML documents, such as *SWI-Prolog* [Wie05] and *CIAO* [CH01].

In this paper, we investigate how to integrate the *XQuery* language and logic programming. With this aim:

1. Following our previous proposal [ABE08, ABE06], an XML document can be seen as a logic program (a Prolog program), by considering *facts* and *rules* for expressing both the XML schema and document.
2. Now, our proposal is that an *XQuery* expression can be translated (i.e. encoded) into logic programming (i.e. into a Prolog program) by introducing *new rules* for the *join* of documents, and for the translation of *for-let-where-return* expressions. Such rules are combined with the rules and facts representing the input XML documents.

3. Finally, a *specific goal* is generated for obtaining the answer of the given *XQuery* expression. From the set of answers of the generated goal, we can rebuild an XML document representing the answer of the given XQuery expression.

In summary, our technique allows the handling of XML documents as follows. Firstly, the input XML documents are loaded. It involves the translation of the XML documents into a logic program. For efficiency reasons, the rules, which correspond to the XML document structure, are loaded in *main memory*, but facts, which represent the values of the XML document, are stored in *secondary memory*, whenever they do not fit in main memory and using appropriate *indexing techniques* [ABE06, ABE08]. Secondly, the user can now write queries against the loaded documents. Each given *XQuery* query is translated into a logic program and a specific goal. The evaluation of such goal takes advantage of the indexing technique to improve the efficiency of query solving. Finally, from the set of answers of the goal, an output XML document can be built. Let us remark that our proposal uses as basis the implementation of *XPath* in logic programming studied in our previous work [ABE08] (for which a bottom-up approach has been also studied in [ABE06]).

The structure of the paper is as follows. Section 2 will present the translation of XML documents into Prolog; Section 3 will review the translation of *XPath* into logic programming; Section 4 will provide the new translation of *XQuery* expressions into logic programming; and finally, Section 5 will conclude and present future work.

2 Translating XML Documents into Logic Programming

In order to define our translation, we need to number the nodes of the XML documents. Similar kinds of node numbering have been studied in some works about XML processing in relational databases [BGvK$^+$05, OOP$^+$04, TVB$^+$02]. Our goal is similar to these approaches: to identify each inner node and leaf of the tree represented by the XML document.

Given an XML document, we can consider a new XML document called *node-numbered XML document* as follows. Starting from the root element numbered as 1, the node-numbered XML document is numbered using an attribute called **nodenumber**[1] where each j-th child of a tagged element is numbered with the sequence of natural numbers $i_1.\ldots.i_t.j$ whenever the parent is numbered as $i_1.\ldots.i_t$: $<tag\ att_1 = v_1, \ldots, att_n = v_n, \mathbf{nodenumber}{=}\ \mathbf{i_1.\ldots.\ i_t.j} >$ $elem_1, \ldots, elem_s < /tag >$. This is the case of tagged elements. If the j-th child is of a basic type (non tagged) and the parent is an inner node, then the element is labeled and numbered as follows: $<$ *unlabeled* $\mathbf{nodenumber} = \mathbf{i_1.\ldots.i_t.j} >$ *elem* $<$ */unlabeled* $>$; otherwise the element is not numbered. It gives to us a *hierarchical and left-to-right numbering* of the nodes of an XML document.

[1] It is supposed that "nodenumber" is not already used as attribute in the tags of the original XML document.

An element in an XML document is further left in the XML tree than another when the node number is smaller w.r.t. the lexicographic order of sequences of natural numbers. Any numbering that identifies each inner node and leaf could be adapted to our translation.

In addition, we have to consider a new document called *type and node-numbered XML document* numbered using an attribute called **typenumber** as follows. Starting the numbering from 1 in the root of the node-numbered XML document, each tagged element is numbered as: $< tag\ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i_1.\ldots, i_t.j, \mathbf{typenumber} = \mathbf{k} > elem_1, \ldots, elem_s < /tag >$. The type number k of the tag is equal to $l + n + 1$ whenever the type number of the parent is l, and n is the number of tagged elements weakly distinct [2] occurring in leftmost positions at the same level of the XML tree [3].

Now, the translation of the XML document into a logic program is as follows. For each inner node in the type and node numbered XML document $< tag\ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i, typenumber = k > elem_1, \ldots, elem_s < /tag >$ we consider the following rule, called *schema rule*:

$$
\begin{array}{ll}
\multicolumn{2}{l}{tag(tagtype(Tag_{i_1}, \ldots, Tag_{i_t}, [Att_1, \ldots, Att_n]), NTag, k, Doc)\text{:-}} \\
& tag_{i_1}(Tag_{i_1}, [NTag_{i_1}|NTag], r, Doc), \\
& \ldots, \\
& tag_{i_t}(Tag_{i_t}, [NTag_{i_t}|NTag], r, Doc), \\
& att_1(Att_1, NTag, r, Doc), \\
& \ldots, \\
& att_n(Att_n, NTag, r, Doc).
\end{array}
$$

where *tagtype* is a new function symbol used for building a Prolog term containing the XML document; $\{tag_{i_1}, \ldots, tag_{i_t}\}$, $i_j \in \{1, \ldots, s\}$, $1 \leq j \leq t$, is the *set of tags* of the tagged elements $elem_1, \ldots, elem_s$; $Tag_{i_1}, \ldots, Tag_{i_t}$ are variables; $att_1, \ldots, att_n$ are the attribute names; $Att_1, \ldots, Att_n$ are variables, one for each attribute name; $NTag_{i_1}, \ldots, NTag_{i_t}$ are variables (used for representing the last number of the node number of the children); $NTag$ is a variable (used for representing the node number of *tag*); k is the type number of *tag*; and finally, r is the type number of the tagged elements $elem_1, \ldots, elem_s$ [4].

In addition, we consider facts of the form: $att_j(v_j, i, k, doc)$ $(1 \leq j \leq n)$, where *doc* is the name of the document. Finally, for each leaf in the type and node numbered XML document: $< tag\ nodenumber = i, typenumber = k > value < /tag >$, we consider the *fact*: $tag(value, i, k, doc)$, where *doc* is the name of the document. For instance, let us consider the following XML document called "books.xml":

[2] Two elements are weakly distinct whenever they have the same tag but not the same structure.

[3] In other words, type numbering is done by levels and in left-to-right order, but each occurrence of weakly distinct elements increases the numbering in one unit.

[4] Let us remark that since *tag* is a tagged element, then $elem_1, \ldots, elem_s$ have been tagged with "unlabeled" labels in the type and node numbered XML document when they were not labeled; thus they must have a type number.

```
<books>
    <book year="2003">
        <author>Abiteboul</author>
        <author>Buneman</author>
        <author>Suciu</author>
        <title>Data on the Web</title>
        <review>A <em>fine</em> book.</review>
    </book>
    <book year="2002">
        <author>Buneman</author>
        <title>XML in Scotland</title>
        <review><em>The <em>best</em> ever!</em></review>
    </book>
</books>
```

Now, the previous XML document can be represented by means of a logic program as follows:

```
Rules (Schema):
———————————————————
books(bookstype(Book, []), NBooks,1,Doc) :-
          book(Book, [NBook|NBooks],2,Doc).
book(booktype(Author, Title, Review, [Year]),
           NBook ,2,Doc) :-
          author(Author, [NAu|NBook],3,Doc),
          title(Title, [NTitle|NBook],3,Doc),
          review(Review, [NRe|NBook],3,Doc),
          year(Year, NBook,3,Doc).
review(reviewtype(Un,Em,[]),NReview,3,Doc):-
     unlabeled(Un,[NUn|NReview],4,Doc),
     em(Em,[NEm|NReview],4,Doc).
review(reviewtype(Em,[]),NReview,3,Doc):-
     em(Em,[NEm|NReview],5,Doc).
em(emtype(Unlabeled,Em,[]),NEms,5,Doc) :-
     unlabeled(Unlabeled,[NUn|NEms],6,Doc),
     em(Em, [NEm|NEms],6,Doc).
```

```
Facts (Document):
———————————————————
year('2003', [1, 1], 3, "books.xml").
author('Abiteboul', [1, 1, 1], 3, "books.xml").
author('Buneman', [2,1, 1], 3, "books.xml").
author('Suciu', [3,1,1], 3, "books.xml").
title('Data on the Web', [4, 1, 1], 3, "books.xml").
unlabeled('A', [1, 5, 1, 1], 4, "books.xml").
em('fine', [2, 5, 1, 1], 4, "books.xml").

   unlabeled('book.', [3, 5, 1, 1], 4, "books.xml").
   year('2002', [2, 1], 3, "books.xml").
   author('Buneman', [1, 2, 1], 3, "books.xml").
   title('XML in Scotland', [2, 2, 1], 3, "books.xml").
   unlabeled('The', [1, 1, 3, 2, 1], 6, "books.xml").
   em('best', [2, 1, 3, 2, 1], 6, "books.xml").
   unlabeled('ever!', [3, 1, 3, 2, 1], 6, "books.xml").
```

Here we can see the translation of each tag into a predicate name: *books*, *book*, etc. Each predicate has four arguments, the first one, used for representing the XML document structure, is encapsulated into a function symbol with the same name as the tag adding the suffix *type*. Therefore, we have *bookstype*, *booktype*, etc. The second argument is used for numbering each node; the third argument of the predicates is used for numbering each type; and the last argument represents the document name. The key element of our translation is to be able to recover the original XML document from the set of rules and facts.

3 Translating XPath into Logic Programming

In this section, we present how *XPath* expressions can be translated into a logic program. Here we present the basic ideas, a more detailed description can be found in [ABE08].

We restrict ourselves to *XPath* expressions of the form $xpathexpr = /expr_1 \ldots /expr_n$ where each $expr_i$ ($1 \leq i \leq n$) can be a tag or a tag with a *boolean condition* of the form [$xpathexpr = value$], where *value* has a basic type. More complex *XPath* queries [W3C07a] can be expressed in *XQuery*, and therefore this restriction does not reduce the expressivity power of our query language.

With the previous assumption, each *XPath* expression $xpathexpr = /expr_1 \ldots /expr_n$ defines a *free of equalities XPath expression*, denoted by $FE(xpathexpr)$. Basically, boolean conditions $[xpathexpr = value]$ are replaced by $[xpathexpr]$ in free of equalities *XPath* expressions. These free of equalities *XPath* expressions define a subtree of the XML document, in which is required that some paths exist (occurrences of boolean conditions $[xpathexpr]$).

For instance, with respect to the *XPath* expression $/books/book\ [author = Suciu]/title$, the free of equalities *XPath* expression is $/books/book\ [author]\ /title$ and the subtree of the type and node numbered XML document which corresponds with the expression $/books/book\ [author]/title$ is as follows:

```
<books nodenumber=1, typenumber=1>
<book year="2003", nodenumber=1.1, typenumber=2>
<author nodenumber=1.1.1 typenumber=3>Abiteboul</author>
<author nodenumber=1.1.2 typenumber=3>Buneman</author>
<author nodenumber=1.1.3 typenumber=3>Suciu</author>
<title nodenumber=1.1.4 typenumber=3>Data on the Web</title>
</book>
<book year="2002" nodenumber=1.2, typenumber=2>
<author nodenumber=1.2.1 typenumber=3>Buneman</author>
<title nodenumber=1.2.2 typenumber=3>XML in Scotland</title>
</book>
</books>
```

Now, given a type and node numbered XML document $\mathcal{D}$, a program $\mathcal{P}$ representing $\mathcal{D}$, and an *XPath* expression *xpathexpr* then the *logic program representing xpathexpr* is $\mathcal{P}^{xpathexpr}$, obtained from $\mathcal{P}$ taking the schema rules for the subtree of $\mathcal{D}$ defined by $FE(xpathexpr)$, and the facts of $\mathcal{P}$. For instance, with respect to the above example, the schema rules defined by $/books/book\ [author]/title$ are:

```
books(bookstype(Book, []), NBooks, 1,Doc):-
        book(Book, [NBook|NBooks], 2,Doc).
book(booktype(Author,Title,Review,[Year]),NBook,2,Doc) :-
        author(Author,[NAuthor|NBook],3,Doc),
        title(Title,[NTitle|NBook],3,Doc).
```

and the facts are the same as the original program. Let us remark that in practice, these rules can be obtained from the schema rules by removing the predicates which do not occur as tags in the free of equalities *XPath* expression. Now, given a type and node numbered XML document, and an *XPath* expression *xpathexpr*, the set of *goals obtained from xpathexpr* are defined as follows.

Firstly, each *XPath* expression *xpathexpr* can be mapped into a set of Prolog terms, denoted by $PT(xpathexpr)$, representing the *patterns* of the query. Due to XML records can have different structure, one pattern is generated for each kind of record. To each pattern t we can associate a set of type numbers, denoted by $TN(t)$.

Now, the *goals* are defined as: $\{tag(Pattern, Node, Type, doc)\{Pattern \rightarrow t, Type \rightarrow r\} \mid t \in PT(xpathexpr), r \in TN(t)\}$ where *tag* is the leftmost tag in *xpathexpr* with a boolean condition; r is a type number associated to each pattern (i.e. $r \in TN(t)$); *Pattern*, *Node* and *Type* are variables; and *doc* is the document name of the input XML document. In the case of *xpathexpr* without boolean conditions we have that *tag* is the rightmost one.

For instance, with respect to $/books/book\ [author = Suciu]/title$, then $PT($ $/books/\ book\ [author = Suciu]/title) = \{booktype('Suciu', Title, Review, [Year])\}$, $TN(booktype('Suciu', Title, Review, [Year])) = \{2\}$, and therefore the (unique) goal is $: -book(booktype('Suciu', Title, Review, Year), Node, 2,''books.xml'')$.

We will call *head tag* of *xpathexpr* to the leftmost tag with a boolean condition, and it will be denoted by $htag(xpathexpr)$. In the case of *xpathexpr* without boolean conditions then the head tag is the rightmost one. In the previous example, $htag(/books/book[author = Suciu]/title) = book$.

In summary, the handling of an *XPath* query involves the *"specialization"* of the schema rules of the XML document (removing predicates) and the *generation* of one or more goals. The goals are obtained from the patterns and the leftmost tag with a boolean condition on the *XPath* expression. Obviously, instead of a set of goals for each *XPath* expression, a unique goal can be considered by adding a new rule. In such a case, the head tag would be the name of the predicate of the added rule.

4 Translating XQuery into Logic Programming

Similarly to *XPath*, *XQuery* expressions can be translated into a logic program generating the corresponding goal. We will focus on a subset of *XQuery*, called *XQuery* core language, whose grammar can be defined as follows.

Core XQuery

$xquery := dxpfree \mid <tag>' \{'xquery, \ldots, xquery'\}' </tag> \mid flwr.$

$dxpfree := document(doc)\ '/'\ xpfree.$

$flwr := \textbf{for}\ \$var\ \textbf{in}\ vxpfree\ [\textbf{where}\ constraint]\ \textbf{return}\ xqvar \mid$

$\qquad \textbf{let}\ \$var\ \textbf{:=}\ vxpfree\ [\textbf{where}\ constraint]\ \textbf{return}\ xqvar.$

$xqvar := vxpfree \mid <tag>' \{'xqvar, \ldots, xqvar'\}' </tag> \mid flwr.$

$vxpfree := \$var \mid \$var\ '/'\ xpfree \mid dxpfree.$

$Op := <= \mid >= \mid < \mid > \mid =.$

$constraint := vxpfree\ Op\ value \mid vxpfree\ Op\ vxpfree$

$\qquad \mid constraint\ '\textbf{or}'\ constraint \mid constraint\ '\textbf{and}'\ constraint.$

where *value* is an XML document, *doc* is a document name, and $xpfree$ is a free of equalities *XPath* expression. Let us remark that *XQuery* expressions use free of equalities *XPath* expressions, given that equalities can be always introduced in *where* expressions. We will say that an *XQuery* expression *ends with attribute name* whenever the *XQuery* expression has the form of an *vxpfree* expression, and the rightmost element has the form *@att*, where *att* is an attribute name. The translation of an *XQuery* expression involves the following steps:

- Firstly, for each *XQuery* expression *xquery*, we can define a logic program $\mathcal{P}^{xquery}$ and a goal.
- Secondly, analogously to *XPath* expressions, for each *XQuery* expression *xquery*, we can define the so-called *head tag*, denoted by $htag(xquery)$, denoting the *predicate name* used for the building of the goal (or subgoal whether the expression *xquery* is nested).

- Finally, for each *XQuery* expression *xquery*, we can define the so-called *tag position*, denoted by *tagpos*(*xquery*), representing the argument of the head tag (i.e. the argument of the predicate) in which the answer is retrieved.

In other words, in the translation each *XQuery* expression can be mapped into a program $\mathcal{P}^{xquery}$ and into a goal of the form : $-tag(\overline{Tag}, Node, Type, Docs)$, where *tag* is the head tag, $\overline{Tag} \equiv Tag_1, \ldots, Tag_n$ are variables, and Tag_{pos} represents the answer of the query, where $pos = tagpos(xquery)$. In addition, *Node* and *Type* are variables representing the node and type numbering of the output document, and *Docs* is a variable representing the documents involved in the query. As a particular case of *XQuery* expressions, *XPath* expressions *xpathexpr* hold that $tagpos(xpathexpr) = 1$.
As running example, let us suppose a query requesting the year and title of the books published before 2003.

```
xquery = for $book in document ('books.xml')/books/book
            return let $year := $book/@year
            where $year<2003
            return <mybook>{$year, $book/title}</mybook>
```

For this query, the translation is as follows:

```
P^xquery = {
(1) mybook(mybooktype(Title, [Year]), [Node], [Type], [Doc]) : −
        join(Title, Year, Node, Type, Doc).

(2) join(Title, Year, [Node], [Type], [Doc]) : −
        vbook(Title, Year, Node, Type, Doc),
        constraints(vbook(Title, Year)).

(3) constraints(Vbook) : −lc(Vbook).
    lc(Vbook) : −c(Vbook).
    c(vbook(Title, Year)) : −le(Year, 2003).

(4) vbook(Title, Year, [Node, Node], [TTitle, TYear], "books.xml") : −
        title(Title, [NTitle|Node], TTitle, "books.xml"),
        year(Year, Node, TYear, "books.xml").
}
```

Basically, the translation of *XQuery* expressions needs to consider the following elements:

- The so-called *document variables* which are *XQuery* variables associated to XML documents by means of *for* or *let* expressions.
- Variables which are not document variables. Each one of these variables can be associated to a document variable. The value of these variables depends on the value of the associated document variable. Such dependence is expressed by means of a *for* or *let* expression. In this case, we say that the associated document variable is the *root* of the given variable.
- *XPath* expressions associated to a document variable. Such *XPath* expressions are those ones such that: (a) the document variable occurs in the *XPath* expression or (b) a variable whose root is the document variable occurs in the *XPath* expression.
- Constraints associated to a document variable. Such constraints are those including *XPath* expressions associated to the given document variable.

In the example, there is only one document variable, that is $\$book$, associated to "books.xml" by means of a *for* expression, and $\$year$ can be associated to $\$book$ whose dependence is expressed by means of the *let* expression. Therefore, $\$book$ is the *root* of $\$year$. In addition, there are two *XPath* expressions associated to $\$book$: $\$year$ and $\$book/title$. Finally, the constraint *"$year<2003"* is associated to the document variable *$book*. Now, the translation of *XQuery* expressions can be summarized as follows:

- The *return* expression generates one or more rules for describing the structure of the output XML document.
- Such structure is generated by means of a special predicate called *join*, defined by means of one rule, whose role is to make the join of multiple documents.
- The predicate *join* calls to predicates called *vvar*'s, one for each *$var*, where *$var* is a document variable.
- Each *vvar* predicate calls to the predicates of the head tags of the *XPath* expressions associated to *$var*.
- The predicate *join* also calls to a special predicate called *constraints*, defined by one rule, whose role is to check the constraints of the *where* expressions included in the *XQuery* expression. The *constraints* predicate calls to $lc^1,\ldots,lc^n$, one for each document variable i $(1 \leq i \leq n)$, and each one of them checks a list of constraints for the given document variable *i*. $c^i_1,\ldots,$ c^i_m check each constraint k $(1 \leq k \leq m)$ of a document variable *i*.

In the example, rule **(1)** defines the structure of the output XML document according to the *return* expression in which a *mybook* record is built including *title* and *year* as attribute. Rule **(2)** of the predicate *join* generates such structure by calling the predicate *vbook* which represents the document variable *$book*. In addition, *join* also calls to the predicate *constraints* by checking the *where* expression. The *vbook* predicate (rule **(4)**) calls to the head tags of the *XPath* expressions associated to the document variable *$book*. In this case, the head tags of $\$year$ and $\$book/title$ are *title* and *year*, respectively. Finally, rule **(3)** declares the special predicate *constraints* which checks the constraint *"$year<2003"* associated to *$book*. Since there is only one document variable and one constraint, the transformation only generates predicates called *lc* and *c*, in order to check the given constraint. The predicate *le* represents the operator "<".

With respect to node and type numbering, we adopt the following convention. The output XML document can be built from several input XML documents. Therefore it is possible that the original numbering is not valid for numbering the output document. However, we can still number the output XML documents by considering as identifier (node and type number) of each record of the output document the list of identifiers (node and type numbers) of the input documents. Such numbering allow to identify each record of the output XML document. This is the reason why rules **(1)**, **(2)** and **(4)** collect in a Prolog list the type and node number of the called predicates (in this case, there is only one input document).

With respect to the goal, the head tag of each $\mathcal{P}^{xquery}$ has to be computed in each case (see next section for more details). In the example, the head tag is *mybook*, that is, $htag(xquery) = mybook$, and the *tagpos* is *1*, that is, $tagpos(xquery) = 1$. Therefore, the goal is : $-mybook(MyBook, Node, Type, Doc)$ and the answer is:

```
MyBook = mybooktype(″XML in Scottland″, [″2002″]), Node = [[[[2, 1], [2, 1]]]]
Type = [[[3, 3]]], Doc = [[″books.xml″]]
```

This answer represents the following XML document:

```
<mybook year=″2002″>
    <title>XML in Scotland</title>
</mybook>
```

In order to build the output XML document from the set of answers, we have to consider some *auxiliary rules* for expressing the schema of the XML output documents. In the example, the schema rules are the following:

```
mybook(mybooktype(Title, [Year]), [[[Node1, Node2]]], [[[Type1, Type2]]], [[Doc]]) : −
        title(Title, [NTitle|Node1], Type1, Doc),
        year(Year, Node2, Type2, Doc).
```

Similarly to input documents, in output XML documents the children are numbered with a larger number than parents. In the example the *mybook* element is numbered as $[[[[2, 1], [2, 1]]]]$ and the child *title* is numbered as $[2, 2, 1]$.

4.1 Formalizing the Transformation

In this section, we show an algorithm for encoding XQuery in logic programming. This algorithm will be illustrated with an example. Assuming the notation of Table 1, the algorithm is shown in Tables 2 and 3. The algorithm has the following elements:

(1) It distinguishes cases for each type of *XQuery* expression;
(2) It defines the values for $\mathcal{P}^{xquery}$, $htag(xquery)$ and $tagpos(xquery)$ in each case;
(3) It uses the notation $\mathcal{P}^{\mathcal{X}}_{\Gamma}$ in order to denote the encoding of a set $\mathcal{X}$ of *XQuery* expressions w.r.t. a context Γ;
(4) The context Γ includes assertions of the form $(\$var, let, xpathexpr, C)$ and $(\$var, for, xpathexpr, C)$ whose meaning is the following: the *XQuery* variable $\$var$ has been assigned to *xpathexpr* by means of a *let* (resp. a *for*) expression with the list of constraints C.

The most relevant cases of the algorithm are cases **(2)** of Table 2, and **(8)** of Table 3.

Case **(2)** introduces the rule for providing structure to the output document. The set $\{tag_1, \ldots, tag_k, att_1, \ldots, att_s\}$ contains the head tags of the expressions $xquery_1, \ldots, xquery_n$, and for each one of them, the type and node numbers are collected in a Prolog list. In addition, the tag position allows to know which arguments have to be selected from the call to the head tags (it is expressed in the conditions of case **(2)**).

Table 1. Notation

$Vars(\Gamma) =_{def} \{\$var \mid (\$var, let, vxpfree, C) \in \Gamma \ or \ (\$var, for, vxpfree, C) \in \Gamma\}$; Denotes the variables of a context Γ;
$DocVars(\Gamma) =_{def} \{\$var \mid (\$var, let, dxpfree, C) \in \Gamma \ or \ (\$var, for, dxpfree, C) \in \Gamma\}$; Denotes the document variables of a context Γ;
$Doc(\$var, \Gamma) =_{def} doc$ whenever $\overline{\Gamma}_{\$var} = document(doc)/xpfree$; Denotes the document associated to a document variable $\$var$ in a context Γ;
$\Gamma_{\$var} =_{def} vxpfree$ whenever $(\$var, let, \ vxpfree, C)$ or $(\$var, for, \ vxpfree, C) \in \Gamma$; Denotes the *XPath* expression associated to a variable $\$var$ in a context Γ;
$\overline{\Gamma}_{\$var} =_{def} vxpfree[\lambda_1 \cdot \ldots \cdot \lambda_n]$ where $\lambda_i = \{\$var_i \rightarrow \Gamma_{\$var_i}\}$ and $\{\$var_1, \ldots, \$var_n\} = Vars(\Gamma)$; Denotes the free of variables *XPath* expression associated to a variable $\$var$ in a context Γ; Variables are replaced by the associated *XPath* expression;
$Root(\$var) =_{def} \$var'$ whenever $\$var \in DocVars(\Gamma)$ and $\$var = \$var'$ or $((\$var, let, \$var''/xpfree, C) \in \Gamma \ or \ (\$var, for, \$var''/xpfree, C) \in \Gamma$ and $Root(\$var'') = \$var')$; Denotes the root of a given variable $\$var$;
$Rootedby(\$var, \mathcal{X}) =_{def} \{xpfree \mid \$var/xpfree \in \mathcal{X}\}$; Denotes the *XPath* expression associated to $\$var$ in $\mathcal{X}$;
$Rootedby(\$var, \Gamma) =_{def} \{xpfree \mid \$var/xpfree \ Op \ \ vxpfree \in C$ or $\$var/xpfree \ \ Op \ value \in C, \ C \in Constraints(\$var, \Gamma)\}$; Denotes the *XPath* expression associated to $\$var$ in a context Γ;
$Constraints(\$var, \ \Gamma) =_{def} \{C_i \mid 1 \leq i \leq n, \ C \equiv C_1 \ Op \ \ldots \ Op \ C_n,$ $(\$var, let, vxpfree, C) \in \Gamma \ or \ (\$var, for, vxpfree, \ C) \in \Gamma\}$ Denotes the list of constraints associated to $\$var$ in a context Γ;

Case **(8)** properly introduces the rule of *join*, which calls *vvar* predicates for each document variable $\$var$. In addition, the *join* predicate calls to the *constraints* predicate. The tag position indicates the argument to be selected from the call to the *vvar* predicate (condition **(b)**). Each *vvar* predicate calls to the head tags of the *XPath* expressions associated to the document variable $\$var$ (condition **(c)**). Finally, the *constraints* predicate calls to predicates $lc^1, \ldots, lc^n$ which check each constraint in a sequential way if the connective is **and**, and otherwise, the algorithm introduces alternative rules for each **or** connective (conditions **(e)** and **(f)**).

In the running example, case **(2)** is applied to $< mybook > \$year, \$book/title < /mybook >$, and the head tags of $\$year$ and $\$book/title$ are *join*. For this reason the rule **(1)** of the running example has the form $mybook(\ldots) : - join(\ldots)$. Case **(8)** is applied to *vbook*, calling the predicates *title* and *year* which are the head tags of the associated *XPath* expressions $\$year$ and $\$book/title$. Finally, the *constraints* predicate calls to *lc*, which at the same time calls to *c* for checking the constraint $\$year < 2003$.

As an example of application of the algorithm, let us suppose the following *XQuery* expression:

Table 2. Translation of XQuery into Logic Programming

(1) $\mathcal{P}^{document(doc)/xpfree} =_{def} \mathcal{P}^{xpfree}$ $htag(document(doc)/xpfree) =_{def} htag(xpfree)$ $tagpos(document(doc)/xpfree) =_{def} tagpos(xpfree)$	
(2) $\mathcal{P}^{<tag>\{xquery_1,\ldots,xquery_n\}</tag>} =_{def}$ $\{\mathcal{R}\} \cup_{1\leq i\leq n} \mathcal{P}^{xquery_i}$ *and* $\mathcal{R} \equiv$ $tag(tagtype(Tag^1_{p_1},\ldots,Tag^k_{p_k},[Att^1_{q_1},\ldots,Att^s_{q_s}]),$ $[NTag_1,\ldots,NTag_k,NAtt_1,\ldots,NAtt_s],$ $[TTag_1,\ldots,TTag_k,TAtt_1,\ldots,TAtt_s],$ $[DTag_1,\ldots,DTag_k,DAtt_1,\ldots,DAtt_s]) :-$ $tag_1(\overline{Tag^1}, NTag_1, TTag_1, DTag_1),$ $\ldots$ $tag_k(\overline{Tag^k}, NTag_k, TTag_k, DTag_k),$ $att_1(\overline{Att^1}, NAtt_1, TAtt_1, DAtt_1),$ $\ldots$ $att_s(\overline{Att^s}, NAtt_s, TAtt_s, DAtt_s).$ $htag(xquery) =_{def} tag,\ tagpos(xquery) =_{def} 1$	– $\overline{Tag^t}\ 1 \leq t \leq k,$ *denotes* $Tag^t_1,\ldots,Tag^t_r$ *where* r *is the arity of* tag_t; – $\overline{Att^j}\ 1 \leq j \leq s,$ *denotes* $Att^j_1,\ldots,Att^j_s$ *where* s *is the arity of* att_j; – *for every* $j \in \{1,\ldots,n\}$ $htag(xquery_j) = att_i,$ $tagpos(xquery_j) = q_i,$ $1 \leq i \leq s,$ *whenever* $xquery_j$ *ends with attribute names,* *and* $htag(xquery_j) = tag_t,$ $tagpos(xquery_j) = p_t$ $1 \leq t \leq k,$ *otherwise*
(3) $\mathcal{P}^{\mathbf{for}\ \$var\ \mathbf{in}\ vxpfree\ [\mathbf{where}\ C]\ \mathbf{return}\ xqvar} =_{def}$ $\mathcal{P}^{xqvar}_{\{(\$var,for,vxpfree,C)\}}$ $htag\ (xquery) =_{def} htag(xqvar)$ $tagpos(xquery) =_{def} tagpos(xqvar)$	
(4) $\mathcal{P}^{\mathbf{let}\ \$var\ :=\ vxpfree\ [\mathbf{where}\ C]\ \mathbf{return}\ xqvar} =_{def}$ $\mathcal{P}^{xqvar}_{\{(\$var,let,vxpfree,C)\}}$ $htag\ (xquery) =_{def} htag(xqvar)$ $tagpos(xquery) =_{def} tagpos(xqvar)$	
(5) $\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def}$ $\{\mathcal{R}\}\cup\ \mathcal{P}_{\Gamma}^{\mathcal{X}-\{xquery/xpfree\}\cup_{1\leq i\leq n}\{xqvar_i/xpfree_0\}}$ *and* $\mathcal{R} \equiv$ $tag(tagtype(Tag_1,\ldots,Tag_r,[Att^1,\ldots,Att^m]),$ $[Node_1,\ldots,Node_s],$ $[Type_1,\ldots,Type_s],$ $[Doc_1,\ldots,Doc_s]) :-$ $tag_1(\overline{Tag^1}, Node_1, Type_1, Doc_1),$ $\ldots$ $tag_s(\overline{Tag^s}, Node_s, Type_s, Doc_s).$ $htag(xquery/xpfree) =_{def} tag$ $tagpos(xquery/xpfree) =_{def} 1$	– $\overline{Tag}^t\ (1 \leq t \leq s)$ *denotes* $Tag^t_1,\ldots,Tag^t_a$ *where* a *is the arity of* tag_t; – $xquery/xpfree \in \mathcal{X}$ *and* $xpfree \equiv /tag/xpfree_0$; – $xquery \equiv <tag> \{xqvar_1,\ldots,$ $xqvar_n\} </tag>$; – $\{tag_1,\ldots,tag_s\} =$ $\{htag\ (xqvar_i\ /xpfree_0) \mid$ $1 \leq i \leq n\}$; – *for every* $p \in \{1,\ldots,n\}$ $Tag_i = Tag^j_{p_j},\ 1 \leq i \leq r,$ *whenever* $tagpos(xqvar_p\ /xpfree_0) = p_j,$ $htag(xqvar_p\ /xpfree_0) = tag_j,$ *and* $Att^l = Tag^j_{p_j},\ 1 \leq l \leq m,$ *whenever* $tagpos(xqvar_p\ /xpfree_0) = p_j,$ $htag(xqvar_p\ /xpfree_0) = tag_j$ *and* $xqvar_p\ /xpfree_0$ *ends with attribute names*

(6) $\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \mathcal{P}^{\mathcal{X}-\{xquery/xpfree\}\cup\{xqvar/xpfree\}}_{\Gamma\cup\{(\$var,for,vxpfree,C)\}}$
$htag(xquery/xpfree) =_{def} htag(xqvar/xpfree)$
$tagpos(xquery/xpfree) =_{def} tagpos(xqvar/xpfree)$
	– $xquery/xpfree \in \mathcal{X}$,
	– $xquery \equiv$
	for $\$var$ **in** $vxpfree$ [**where** C]
	return $xqvar$

(7) $\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \mathcal{P}^{\mathcal{X}-\{xquery/xpfree\}\cup\{xqvar/xpfree\}}_{\Gamma\cup\{(\$var,let,vxpfree,C)\}}$ $htag(xquery/xpfree) =_{def} htag(xqvar/xpfree)$ $tagpos(xquery/xpfree) =_{def} tagpos(xqvar/xpfree)$	– $xquery/xpfree \in \mathcal{X}$, – $xquery \equiv$ **let** $\$var$:= $vxpfree$ [**where** C] **return** $xqvar$

Table 3. Translation of XQuery into Logic Programming (cont'd)

(8) $\{\mathcal{J}^{\Gamma}\} \cup \mathcal{C}^{\Gamma} \cup \{\mathcal{R}^{\$var} \mid \$var \in DocVars(\Gamma)\}$ $\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \bigcup_{\substack{\$var \in DocVars(\Gamma),\\ \$var = Root(\$var'),\\ xpfree \in Rootedby(\$var', \mathcal{X}) \cup Rootedby(\$var', \Gamma)}} \mathcal{P}^{\bar{\Gamma}_{\$var'}/xpfree}$ **(a)**	**(a)** – $\mathcal{X}$ *does not include* *tagged elements* *and flwr expressions*
$\mathcal{J}^{\Gamma} \equiv$ $join(Tag_1, \ldots, Tag_m, [Node_1, \ldots, Node_n],$ $[Type_1, \ldots, Type_n], [Doc_1, \ldots, Doc_n]) :-$ $vvar_1(\overline{Tag^1}, Node_1, Type_1, Doc_1),$ **(b)** $\ldots$ $vvar_n(\overline{Tag^n}, Node_n, Type_n, Doc_n),$ $constraints(vvar_1(\overline{Tag^1}), \ldots, vvar_n(\overline{Tag^n})).$	**(b)** – $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma);$ – *for each* $\$var'/xpfree_j \in \mathcal{X}$ *such that* $Root(\$var') = \$var_i$ *and* $tagpos(\overline{\Gamma}_{\$var'}/xpfree_j) = p_j$ *then* $Tag_j = Tag^i_{p_j}$ – $\overline{Tag^i} = Tag^i_1 \ldots Tag^i_s$ *one* Tag^i_r, $1 \leq r \leq s$ *for each* $\$var'/xpfree_r \in \mathcal{X} \cup \Gamma$ *such that* $Root(\$var') = \$var_i$
$\mathcal{R}^{\$var} \equiv$ $vvar(Tag_1, \ldots, Tag_n, Node, [Type_1, \ldots, Type_n], doc) :-$ $tag_1(Tag_1, [Node_{11}, \ldots, Node_{1k_1} \mid NTag], Type_1, doc),$ **(c)** $\ldots,$ $tag_n(Tag_n, [Node_{n1}, \ldots, Node_{nk_n} \mid NTag], Type_n, doc).$	**(c)** – $doc = Doc(\$var, \Gamma)$ – $tag_i = htag(\overline{\Gamma}_{\$var'}/xpfree)$ $\$var'\ /xpfree \in \mathcal{X}$ $\$var = Root(\$var')$ – $Node = [N_1, \ldots, N_n]$ *and* $N_i = [Node_{ik_i} \mid NTag]$ *if* $(\$var', for, vxpfree, C) \in \Gamma$, *and* $N_i = NTag$, *otherwise*
$\mathcal{C}^{\Gamma} \equiv \{$ $constraints(Vvar_1, \ldots, Vvar_n) :-$ $lc^1_1(Vvar_1, \ldots, Vvar_n),$ $\ldots$ **(d)** $lc^n_1(Vvar_1, \ldots, Vvar_n).$ $\} \cup_{\$var \in Vars(\Gamma), C^j \in constraints(\$var, \Gamma)} \mathcal{C}^j$	**(d)** $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$
$\mathcal{C}^j \equiv$ $\{lc^j_i(Vvar_1, \ldots, Vvar_n) :-$ $c^j_i(Vvar_1, \ldots, Vvar_n), lc^j_{i+1}(Vvar_1, \ldots, Vvar_n).$ $\mid 1 \leq i \leq n, Op_i = \textbf{and}\}$ $\cup$ **(e)** $\{lc^j_i(Vvar_1, \ldots, Vvar_n) :- c^j_i(Vvar_1, \ldots, Vvar_n).$ $lc^j_i(Vvar_1, \ldots, Vvar_n) :- lc^j_{i+1}(Vvar_1, \ldots, Vvar_n).$ $\mid 1 \leq i \leq n, Op_i = \textbf{or}\}$ $\cup_{\{c^j_i \mid 1 \leq i \leq n\}} \{\mathcal{C}^j_i\}$	**(e)** – $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma),$ – $C^j \equiv c^j_1 Op_1 \ldots, Op_n c^j_n$
$\mathcal{C}^j_i \equiv c^j_i(vvar_1(\overline{Tag^1}), \ldots, vvar_n(\overline{Tag^n})) :- Op(Tag^k_j, value).$ **(*)** $\mathcal{C}^j_i \equiv c^j_i(vvar(\overline{Tag^1}), \ldots, vvar(\overline{Tag^n})) :- Op(Tag^k_j, Tag^m_r).$ **(**)** **(f)**	**(f)** – $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$ – **(*)** $c^j_i \equiv \$var'/xpfree_j$ *Op value* *and* $Root(\$var') = \$var_k$ – **(**)** $c^j_i \equiv \$var'/xpfree_j$ *Op* $\$var'/xpfree_r,$ $Root(\$var') = \$var_k$ *and* $Root(\$var') = \$var_m$
$htag(\$var/xpfree_j) =_{def} join$ $tagpos(\$var/xpfree_j) =_{def} j$ **(g)**	**(g)** *for every* $\$var \in Vars(\Gamma)$, $xpfree_j \in Rootedby(\$var, \mathcal{X}) \cup Rootedby(\$var, \Gamma)$

```
xquery=
Let $store1 := document ("books1.xml")/books
  $store2 := document("books2.xml")/books
   return
    for $book1 in $store1/book
     $book2 in $store2/book
      return
       let $title := $book1/title
        where $book1/@year<2003 and $title=$book2/title
          return <mybook>{
           $title,
           $book1/review,
           $book2/review
           }
           </mybook>
```

requesting the reviews of books (published before 2003) occurring in two documents: the first one is the running example and the second one is:

```
<books>
  <book year="2003">
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
    <title>Data on the Web</title>
    <review>very good</review>
  </book>
  <book year="2002">
    <author>Buneman</author>
    <title>XML in Scotland</title>
    <review>Good reference!</review>
  </book>
</books>
```

In this case, the *return* expression generates a new rule *mybook* in which the *title* is obtained from the first document and *review*'s are obtained from both documents. The application of the algorithm is as follows:

$$\begin{array}{l}
\mathcal{P}^{xquery} =_{(Rule(\mathbf{4}))} \mathcal{P}^{xquery_1}_{(\$store1, let, document(''books1.xml'')/books, \emptyset)} =_{(Rule(\mathbf{4}))} \\
\mathcal{P}^{xquery_2}_{\Gamma_1} =_{(Rule(\mathbf{3}))} \mathcal{P}^{xquery_3}_{\Gamma_1 \cup \{(\$book1, for, \$store1, \emptyset)\}} =_{(Rule(\mathbf{3}))} \\
\mathcal{P}^{xquery_4}_{\Gamma_1 \cup \{(\$book1, for, \$store1, \emptyset), (\$book2, for, \$store2, \emptyset)\}} =_{(Rule(\mathbf{4}))} \\
\mathcal{P}^{xquery_5}_{\Gamma_2} =_{(Rule(\mathbf{2}))} \{\mathcal{R}\} \cup \mathcal{P}^{\$title, \$book1/review, \$book2/review}_{\Gamma_2}
\end{array}$$

where $\Gamma_1 = \{(\$store1, let, document(''books1.xml'')/books, \emptyset), (\$store2, let, document\ (''books2.xml'')/\ books, \emptyset)\}$ and also $\Gamma_2 = \Gamma_1 \cup \{(\$book1, for, \$store1, \emptyset), (\$book2, for, \$store2, \emptyset), (\$title, let, \$book1/\ title, \$book1/\ @year < 2003\ and\ \$title = \$book2/title)\}$. In addition, $\mathcal{R}$ is defined as follows:

$$\mathcal{R} = \begin{array}{c} mybook(mybooktype(Title, Review1, Review2, []), [Node], [Type], [Doc]) :- \\ join(Title, Review1, Review2, Node, Type, Doc). \end{array}$$

where $join = htag(\$title)$, $join = htag(\$book1/review)$, $join = htag(\$book2/review)$, $tagpos\ (\$title) = 1$, $tagpos\ (\$book1/review) = 2$, $tagpos(\$book2/review) = 3$.
Now, $\mathcal{P}^{\$title, \$book1/review, \$book2/review}_{\Gamma_2}$ is defined as:

$$\begin{array}{l}
\mathcal{P}^{\$title, \$book1/review, \$book2/review}_{\Gamma_2} =_{Rule(\mathbf{8})} \\
\qquad \{\mathcal{J}^{\Gamma}\} \cup \mathcal{C}^{\Gamma} \cup \{\mathcal{R}^{\$store1}, \mathcal{R}^{\$store2}\} \cup \\
\qquad \mathcal{P}^{document(books1.xml)/books/book/title} \cup \\
\qquad \mathcal{P}^{document(books1.xml)/books/book/@year} \cup \\
\qquad \mathcal{P}^{document(books1.xml)/books/book/review} \cup \\
\qquad \mathcal{P}^{document(books2.xml)/books/book/title} \cup \\
\qquad \mathcal{P}^{document(books2.xml)/books/book/review}
\end{array}$$

where $\mathcal{J}^{\Gamma}$ and $\mathcal{C}^{\Gamma}$ are defined as follows:

$$\mathcal{J}^{\Gamma} = \begin{array}{l} join(Title1, Review1, Review2, [Node1, Node2], [Type1, Type2], [Doc1, Doc2]) :- \\ \qquad vstore1(Title1, Year1, Review1, Node1, Type1, Doc1), \\ \qquad vstore2(Title2, Review2, Node2, Type2, Doc2), \\ \qquad constraints(vstore1(Title1, Year1, Review1), vstore2(Title2, Review2)). \end{array}$$

$$\begin{array}{l}
\mathcal{C}^{\Gamma} = \{ \\
\;constraints(Vstore1, Vstore2) :- \\
\qquad\quad lc^{1}(Vstore1, Vstore2). \\
lc^{1}_{1}(Vstore1, Vstore2) :- c^{1}_{1}(Vstore1, Vstore2), \\
\qquad\quad c^{1}_{2}(Vstore1, Vstore2). \\
\;c^{1}_{1}(vstore1(Title1, Year1, Review1), vstore2(Title2, Review2)) :- \\
\qquad\quad le(Year1, 2003). \\
\;c^{1}_{2}(vstore1(Title1, Year1, Review1), vstore2(Title2, Review2)) :- \\
\qquad\quad eq(Title1, Title2). \\
\}
\end{array}$$

where

- $DocVars(\Gamma) = \{\$vstore1, \$vstore2\}$,
- $\$title$, $\$book1\ /review$, $\$book2\ /review \in \mathcal{X}$,
- $Root(\$title) = \$vstore1$, $Root(\$book1) = \$vstore1$ and $Root(\$book2) = \$vstore2$,
- $\$book1/@year$ and $\$book2/title$ occur in Γ,
- $Root(\$book1) = \$vstore1$ and $Root(\$book2) = \$vstore2$,
- $C^{1} \equiv c^{1}_{1}$ **and** $c^{1}_{2} \in \Gamma$, $c^{1}_{1} \equiv \$book1/@year < 2003$, $c^{1}_{2} \equiv \$title = \$book2/title$,
- $Root(\$book1) = \$vstore1$, $Root(\$title) = \$vstore1$ and $Root(\$book2) = \$vstore2$

Finally, $\mathcal{R}^{\$store1}$ and $\mathcal{R}^{\$store2}$ are defined as:

$$\begin{array}{l}
\mathcal{R}^{\$store1} = \\
\;vstore1(Title, Year, Review, [Node, Node, Node], [Type_{1}, Type_{2}, Type_{3}], \\
\qquad\quad ''books1.xml'') :- \\
\qquad\quad title(Title, [Node_{1}, Node_{2}|Node], Type_{1}, ''\,books1.xml''), \\
\qquad\quad year(Year, [Node_{2}|Node], Type_{2}, ''\,books1.xml''), \\
\qquad\quad review(Review, [Node_{1}, Node_{2}|Node], Type_{3}, ''\,books1.xml''). \\
\mathcal{R}^{\$store2} = \\
\;vstore2(Title, Review, [Node, Node], [Type_{1}, Type_{2}], ''\,books2.xml'') :- \\
\qquad\quad title(Title, [Node_{1}, Node_{2}|Node], Type_{1}, ''\,books2.xml''), \\
\qquad\quad review(Review, [Node_{1}, Node_{2}|Node], Type_{2}, ''\,books2.xml'').
\end{array}$$

and

$$\begin{array}{l}
\mathcal{P}^{document(books1.xml)/books/book/title} = Facts\ \ of\ \ \mathcal{P} \\
\mathcal{P}^{document(books1.xml)/books/book/@year} = Facts\ \ of\ \ \mathcal{P} \\
\mathcal{P}^{document(books2.xml)/books/book/title} = Facts\ \ of\ \ \mathcal{P} \\
\mathcal{P}^{document(books1.xml)/books/book/review} = \\
\mathcal{P}^{document(books2.xml)/books/book/review} = \\
\{ \\
\;review(reviewtype(Unlabeled, Em, []), NReview, 3, Doc) :- \\
\qquad\quad unlabeled(Unlabeled, [NUnlabeled|NReview], 4, Doc), \\
\qquad\quad em(Em, [NEm|NReview], 4, Doc). \\
\;review(reviewtype(Em, []), NReview, 3, Doc) :- \\
\qquad\quad em(Em, [NEm|NReview], 5, Doc). \\
\;em(emtype(Unlabeled, Em, []), NEms, 5, Doc) :- \\
\qquad\quad unlabeled(Unlabeled, [NUnlabeled|NEms], 6, Doc), \\
\qquad\quad em(Em, [NEm|NEms], 6, Doc). \\
\} \cup Facts\ \ of\ \ \mathcal{P}
\end{array}$$

where

- $''books1.xml'' = Doc(\$vstore1, \Gamma), ''\,books2.xml'' = Doc(\$vstore2, \Gamma)$
- $htag(document\ (''books1.xml'')\ /books/book\ /title) = title$
- $htag(document\ (''books1.xml'')\ /books/book\ /year) = year$
- $htag(document\ (''books1.xml'')\ /books/book\ /review) = review$
- $\overline{\Gamma}_{\$title} = document\ (''books1.xml'')\ /books/book/$
- $\overline{\Gamma}_{\$book1} = document\ (''books1.xml'')\ /books/book/$
- $\overline{\Gamma}_{\$book2} = document\ (''books2.xml'')\ /books/book/$

5 Conclusions and Future Work

In this paper, we have studied how to encode *XQuery* expressions into logic programming. It allows us to evaluate *XQuery* expressions against XML documents using logic rules.

As far as we know, this is the first time that *XQuery* is implemented in logic programming. Previous proposals in this research area are mainly focused on the definition of *new* query languages of logic style [SB02, CF07, May04, Sei02] and functional style [HP03, BCF05] for XML documents, and the only proposal for *XQuery* implementation takes as host language a functional language (i.e. *OCAML*). The proposals of new query languages in this framework have to adapt the unification in the case of logic languages [BS02b, May04, CF03], and the pattern matching in the case of functional languages [BCF05, HP03] in order to accommodate the handling of XML records. However, in our case, we can adopt standard term unification by encoding XML documents into logic programming, and therefore one of the advantages of our approach is that it can be integrated with any Prolog implementation. In addition, the advantage of a logic-based implementation of *XQuery* is that, *XQuery* can be combined with logic programs. Logic programs can be used, for instance, for representing RDF and OWL documents (see, for instance, [WSW03, Wol04]), and therefore XML querying and processing can be combined with RDF and OWL reasoning in our framework –in fact, we have been recently working in a proposal in this line [Alm08].

On the other hand, the proposal of this paper also contributes to the study of the representation and handling of XML documents in relational database systems. In our framework, logic programs represent XML documents by means of rules and a table of facts. In addition, the table of facts is indexed in secondary memory for improving the retrieval. Similar processing and storing can be found in the proposals of XML processing with relational databases (see [BGvK$^+$05], [OOP$^+$04] and [TVB$^+$02]). In fact, we plan to implement the storing of facts in a relational database management system in order to improve fact storing and retrieval.

Therefore our proposal of a *logic-based query language for the Semantic Web* combines the advantages of efficient retrieval of facts in a relational database style together with reasoning capabilities of logic programming.

As future work we would like to implement our technique. We have already implemented *XPath* in logic programming (see `http://indalog.ual.es/Xindalog`). Taking as basis this implementation we would like to extend it to *XQuery* expressions.

References

[ABE06] Almendros-Jiménez, J.M., Becerra-Terón, A., Enciso-Baños, F.J.: Magic sets for the XPath language. Journal of Universal Computer Science 12(11), 1651–1678 (2006)

[ABE08] Almendros-Jiménez, J.M., Becerra-Terón, A., Enciso-Baños, F.J.: Querying XML documents in logic programming. Theory and Practice of Logic Programming 8(3), 323–361 (2008)

[ACJ04] Atanassow, F., Clarke, D., Jeuring, J.: UUXML: A Type-Preserving XML Schema Haskell Data Binding. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 71–85. Springer, Heidelberg (2004)

[Alm08] Almendros-Jiménez, J.M.: An RDF Query Language based on Logic Programming. In: Proceedings of the 3rd Int'l. Workshop on Automated Specification and Verification of Web Systems. Electronic Notes on Theoretical Computer Science, vol. 200, pp. 67–85 (2008)

[BBFS05] Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and Semantic Web Query Languages: A Survey. In: Eisinger, N., Małuszyński, J. (eds.) Reasoning Web. LNCS, vol. 3564, pp. 35–133. Springer, Heidelberg (2005)

[BCF05] Benzaken, V., Castagna, G., Frish, A.: CDuce: an XML-centric general-purpose language. In: Proc. of the ACM SIGPLAN International Conference on Functional Programming, pp. 51–63. ACM Press, New York (2005)

[BCM05] Benzaken, V., Castagna, G., Miachon, C.: A full pattern-based paradigm for XML query processing. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 235–252. Springer, Heidelberg (2005)

[BFG01] Baumgartner, R., Flesca, S., Gottlob, G.: The elog web extraction language. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS, vol. 2250, pp. 548–560. Springer, Heidelberg (2001)

[BGvK$^+$05] Boncz, P.A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: Pathfinder: XQuery - The Relational Way. In: Proc. of the International Conference on Very Large Databases, pp. 1322–1325. ACM Press, New York (2005)

[BHL01] Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web – A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. In: Scientific American, 36 pages (May 2001)

[Bol00a] Boley, H.: Relationships Between Logic Programming and RDF. In: Kowalczyk, R., et al. (eds.) PRICAI-WS 2000. LNCS, vol. 2112, pp. 201–218. Springer, Heidelberg (2001)

[Bol00b] Boley, H.: Relationships between logic programming and XML. In: Proc. of the Workshop on Logic Programming, pp. 19–34, Würzburg, Germany (2000) (GMD Report 90)

[Bol01] Boley, H.: The rule markup language: RDF-XML data model, XML schema hierarchy, and XSL transformations. In: Bartenstein, O., Geske, U., Hannebauer, M., Yoshie, O. (eds.) INAP 2001. LNCS, vol. 2543, pp. 124–139. Springer, Heidelberg (2003)

[BS02a] Bry, F., Schaffert, S.: The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In: Chaudhri, A.B., et al. (eds.) NODe-WS 2002. LNCS, vol. 2593, pp. 295–310. Springer, Heidelberg (2003)

[BS02b] Bry, F., Schaffert, S.: Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 255–270. Springer, Heidelberg (2002)

[CDF$^+$04] Chamberlin, D., Draper, D., Fernández, M., Kay, M., Robie, J., Rys, M., Simeon, J., Tivy, J., Wadler, P.: XQuery from the Experts. Addison-Wesley, Reading (2004)

[CF03] Coelho, J., Florido, M.: Type-based XML Processing in Logic Programming. In: Dahl, V., Wadler, P. (eds.) PADL 2003. LNCS, vol. 2562, pp. 273–285. Springer, Heidelberg (2002)

[CF04] Coelho, J., Florido, M.: CLP(Flex): Constraint logic programming applied to XML processing. In: Meersman, R., Tari, Z. (eds.) OTM 2004. LNCS, vol. 3291, pp. 1098–1112. Springer, Heidelberg (2004)

[CF07] Coelho, J., Florido, M.: XCentric: logic programming for XML processing. In: WIDM 2007: Proceedings of the 9th annual ACM international workshop on Web information and data management, pp. 1–8. ACM Press, New York (2007)

[CH01] Cabeza, D., Hermenegildo, M.: Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. Theory and Practice of Logic Programming 1(3), 251–282 (2001)

[Cha02] Chamberlin, D.: XQuery: An XML Query Language. IBM Systems Journal 41(4), 597–615 (2002)

[HP03] Hosoya, H., Pierce, B.C.: XDuce: A Statically Typed XML Processing Language. ACM Transactions on Internet Technology 3(2), 117–148 (2003)

[May04] May, W.: XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. Theory and Practice of Logic Programming 4(3), 239–287 (2004)

[MS03] Marian, A., Simeon, J.: Projecting XML Documents. In: Proc. of International Conference on Very Large Databases, Burlington, USA, pp. 213–224. Morgan Kaufmann, San Francisco (2003)

[OOP$^+$04] O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: OrdPaths: Insert-friendly XML Node Labels. In: Proc. of the ACM SIGMOD Conference, pp. 903–908. ACM Press, New York (2004)

[SB02] Schaffert, S., Bry, F.: A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In: Proc. of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Aachen, Germany, CEUR Workshop Proceedings 60, 22 pages (2002)

[Sei02] Seipel, D.: Processing XML-Documents in Prolog. In: Procs. of the Workshop on Logic Programming, 15 pages, Dresden, Germany, Technische Universität Dresden (2002)

[Thi02] Thiemann, P.: A typed representation for HTML and XML documents in Haskell. Journal of Functional Programming 12(4&5), 435–468 (2002)

[TVB$^+$02] Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and Querying Ordered XML using a Relational Database System. In: Proc. of the ACM SIGMOD Conference, pp. 204–215. ACM Press, New York (2002)

[W3C07a] W3C. XML Path Language (XPath) 2.0. Technical report (2007), `http://www.w3.org/TR/xpath`

[W3C07b] W3C. XML Query Working Group and XSL Working Group, XQuery 1.0: An XML Query Language. Technical report (2007), `http://www.w3.org`

[Wad02] Wadler, P.: XQuery: A Typed Functional Language for Querying XML. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 188–212. Springer, Heidelberg (2003)

[Wie05] Wielemaker, J.: SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam (March 2005)

[Wol04] Wolz, R.: Web Ontology Reasoning with Logic Databases. PhD thesis, Universität Fridericiana zu Karlsruhe (2004)

[WR99] Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: Proceedings of the International Conference on Functional Programming, pp. 148–159. ACM Press, New York (1999)

[WSW03] Wielemaker, J., Schreiber, G., Wielinga, B.J.: Prolog-Based Infrastructure for RDF: Scalability and Performance. In: Fensel, D., Sycara, K.P., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 644–658. Springer, Heidelberg (2003)

Causal Subgroup Analysis for Detecting Confounding

Martin Atzmueller and Frank Puppe

University of Würzburg,
Department of Computer Science VI
Am Hubland, 97074 Würzburg, Germany
{atzmueller,puppe}@informatik.uni-wuerzburg.de

Abstract. This paper presents a causal subgroup analysis approach for the detection of confounding: We show how to identify (causal) relations between subgroups by generating an extended causal subgroup network utilizing background knowledge. Using the links within the network we can identify relations that are potentially confounded by external (confounding) factors. In a semi-automatic approach, the network and the discovered relations are then presented to the user as an intuitive visualization. The applicability and benefit of the presented technique is illustrated by examples from a case-study in the medical domain.

1 Introduction

Subgroup discovery (e.g., [1,2,3,4]) is a powerful approach for explorative and descriptive data mining to obtain an overview of the interesting dependencies between a specific target (dependent) variable and usually many explaining (independent) variables. The interesting subgroups can be defined as subsets of the target population with a (distributional) unusualness concerning a certain property we are interested in: The risk of coronary heart disease (target variable), for example, is significantly higher in the subgroup of smokers with a positive family history than in the general population.

When interpreting and applying the discovered relations, it is often necessary to consider the patterns in a causal context. However, considering an association as having a causal interpretation can often lead to incorrect results, due to the basic tenet of statistical analysis that association does not imply causation (cf. [5]): A subgroup may not be causal for the target group, and thus can be suppressed by other causal groups. Then, the suppressed subgroup itself is not interesting, but the other subgroups are better suited for characterizing the target concept. Furthermore, the estimated *effect*, that is, the quality of the subgroup may be due to associations with other *confounding* factors that were not considered in the quality computation. For instance, the quality of a subgroup may be confounded by other variables that are associated with the independent variables, and are a direct cause of the (dependent) target variable. Then, it is necessary to identify potential confounders, and to measure or to control their influence concerning the subgroup and the target concept. Let us assume, for example, that ice cream consumption and murder rates are highly correlated. However, this does not necessarily mean that ice cream incites murder or that murder increases the demand for ice cream. Instead, both ice cream and murder rates might be joint effects of a common cause or confounding factor, namely, hot weather.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 136–148, 2009.

In this paper, we present a semi-automatic causal subgroup analysis approach for the detection of (true) causal subgroups and potentially confounded and/or effect-modified relations..We use known subgroup patterns as background knowledge that can be incrementally refined: These patterns represent subgroups that are *acausal*, thus having no causes, and subgroup patterns that are known to be directly causally related to other (target) subgroups. Additionally, both concepts can be combined, for example, in the medical domain certain variables such as *Sex* have no causes, and it is known that they are causal risk factors for certain diseases.

Using the patterns contained in the background knowledge, and a set of subgroups for analysis, we can construct a causal net containing relations between the subgroups. This network can be interactively inspected and analyzed by the user: It directly provides a visualization of the (causal) relations between the subgroups, and also provides for a possible explanation of these. By traversing the relations in the network, we can then identify potential confounding.

The rest of the paper is organized as follows: First, we discuss the background of subgroup discovery, the concept of confounding, and basic constraint-based causal analysis methods in Section 2. After that, we present the causal discovery approach utilizing background knowledge for detecting causal and confounding/effect-modified relations in Section 3. Exemplary results of the application of the presented approach are given in Section 4 using data from a fielded system in the medical domain. Finally, Section 5 concludes the paper with a discussion of the presented work.

2 Background

In this section, we first introduce the necessary notions concerning the used knowledge representation, before we define the setting for subgroup discovery. After that, we introduce the concept of confounding, criteria for its identification, and describe basic constraint-based techniques for causal subgroup analysis.

2.1 Basic Definitions

Let Ω_A denote the set of all attributes. For each attribute $a \in \Omega_A$ a range $dom(a)$ of values is defined; $\mathcal{V}_A$ is assumed to be the (universal) set of attribute values of the form $(a = v)$, where $a \in \Omega_A$ is an attribute and $v \in dom(a)$ is an assignable value. We consider nominal attributes only so that numeric attributes need to be discretized accordingly. Let CB be the case base (data set) containing all available cases (instances): A case $c \in CB$ is given by the n-tuple $c = ((a_1 = v_1), (a_2 = v_2), \ldots, (a_n = v_n))$ of $n = |\Omega_A|$ attribute values, $v_i \in dom(a_i)$ for each a_i.

2.2 Subgroup Discovery

The main application areas of subgroup discovery (e.g., [1,2,3,4]) are exploration and descriptive induction, to obtain an overview of the relations between a (dependent) target variable and a set of explaining (independent) variables. As in the *MIDOS* approach [1], we consider subgroups that are, for example, as large as possible, and have the most unusual (distributional) characteristics with respect to the concept of interest given by a binary target variable. Therefore, not necessarily complete relations but also partial relations, that is, (small) subgroups with "interesting" characteristics can be sufficient.

Subgroup discovery mainly relies on the subgroup description language, the quality function, and the search strategy. Often heuristic methods (e.g., [3]) but also efficient exhaustive algorithms (e.g., the SD-Map algorithm [4]) are applied. The description language specifies the individuals belonging to the subgroup. For a common single-relational propositional language a subgroup description can be defined as follows:

Definition 1 (Subgroup Description). *A subgroup description $sd = e_1 \wedge e_2 \wedge \cdots \wedge e_k$ is defined by the conjunction of a set of selectors $e_i = (a_i, V_i)$: Each of these are selections on domains of attributes, $a_i \in \Omega_A, V_i \subseteq dom(a_i)$. We define Ω_E as the set of all possible selectors and Ω_{sd} as the set of all possible subgroup descriptions.*

A quality function measures the interestingness of the subgroups and is used to rank these. Typical quality criteria include the difference in the distribution of the target variable concerning the subgroup and the general population, and the subgroup size.

Definition 2 (Quality Function). *Given a particular target variable $t \in \Omega_E$, a quality function $q : \Omega_{sd} \times \Omega_E \rightarrow R$ is used in order to evaluate a subgroup description $sd \in \Omega_{sd}$, and to rank the discovered subgroups during search.*

Several quality functions were proposed (cf. [1,2,3,4]), for example, the functions q_{BT} and q_{RG} shown below:

$$q_{BT} = \frac{(p - p_0) \cdot \sqrt{n}}{\sqrt{p_0 \cdot (1 - p_0)}} \cdot \sqrt{\frac{N}{N - n}}, \quad q_{RG} = \frac{p - p_0}{p_0 \cdot (1 - p_0)}, n \geq \mathcal{T}_{Supp} ,$$

p denotes is the relative frequency of the target variable in the subgroup, p_0 is the relative frequency of the target variable in the total population, $N = |CB|$ is the size of the total population, and n denotes the size of the subgroup. In contrast to the quality function q_{BT} (the classic binomial test), the quality function q_{RG} only compares the target shares of the subgroup and the total population measuring the *relative gain*. Therefore, a support threshold $\mathcal{T}_{Supp}$ is necessary to discover significant subgroups.

The result of subgroup discovery is a set of subgroups. Since subgroup discovery methods are not necessarily covering algorithms the discovered subgroups can overlap significantly and their estimated quality (effect) might be confounded by external variables. In order to reduce the redundancy of the subgroups and to identify potential confounding factors, methods for causal analysis can then be applied.

2.3 The Concept of Confounding

Confounding can be described as a bias in the estimation of the effect of the subgroup on the target concept due to attributes affecting the target concept that are not contained in the subgroup description [6]. Thus, confounding is caused by a lack of comparability between subgroup and complementary group due to a difference in the distribution of the target concept caused by other factors.

An extreme case for confounding is presented by *Simpson's Paradox* [7,8]: The (positive) effect (association) between a given variable X and a variable T is countered by a negative association given a third factor F, that is, X and T are negatively correlated in the subpopulations defined by the values of F [7]. For binary variables X, T, F this can be formulated as

$$P(T|X) > P(T|\neg X)\,, P(T|X,F) < P(T|\neg X,F)\,, P(T|X,\neg F) < P(T|\neg X,\neg F)\,.$$

The event X increases the probability of T in a given population while it decreases the probability of T in the subpopulations given by the restrictions on F and $\neg F$. For the example shown in Figure 1, let us assume that there is a positive correlation between the event X that describes *people that do not consume soft drinks* and T specifying the diagnosis *diabetes*. This association implies that people not consuming soft drinks are affected more often by diabetes (50% non-soft-drinkers vs. 40% soft-drinkers). However, this is due to age, if older people (given by F) consume soft drinks less often than younger people, and if diabetes occurs more often for older people, inverting the effect.

Combined	T	¬T	∑	Rate (T)
X	25	25	50	50%
¬X	20	30	50	40%
∑	45	55	100	

Restricted on F	T	¬T	∑	Rate (T)
X	24	16	40	60%
¬X	8	2	10	80%
∑	32	18	50	

Restricted on ¬F	T	¬T	∑	Rate (T)
X	1	9	10	10%
¬X	12	28	40	30%
∑	13	37	50	

Fig. 1. Example: Simpson's Paradox

There are three criteria that can be used to identify a **confounding factor** F [6,9], given the factors X contained in a subgroup description and a target concept T:

1. A confounding factor F must be a *cause* for the target concept T, that is, an independent risk factor for a certain disease.
2. The factor F must be associated/correlated with the subgroup (factors) X.
3. A confounding factor F must *not* be (causally) affected by the factors X.

However, these criteria are only necessary but not sufficient to identify confounders. If purely automatic methods are applied for detecting confounding, then these may label some variables as confounders incorrectly, for example, if the real confounders have not been measured, or if their contributions cancel out. Thus, user interaction is rather important for validating confounded relations. Furthermore, the identification of confounding requires causal knowledge since confounding is itself a causal concept [9].

Proxy Factors and Effect Modification There are two phenomena that are closely related to confounding. First, a factor may only be associated with the subgroup but may be the real cause for the target concept. Then, the subgroup is only a **proxy factor**. Another situation is given by **effect modification**: Then, a third factor F does not necessarily need to be associated with the subgroup described by the factors X; F can be an additional factor that increases the effect of X in a certain subpopulation only, pointing to new subgroup descriptions that are interesting by themselves.

2.4 Constraint-Based Methods for Causal Subgroup Analysis

In general, the philosophical concept of causality refers to the set of all particular 'cause-and-effect' or 'causal' relations. A subgroup is causal for the target group, if in an ideal experiment [5] the probability of an object not belonging to the subgroup to be a member of the target group increases or decreases when the characteristics of the object are changed such that the object becomes a member of the subgroup. For example, the probability that a patient survives (target group) increases if the patient received a special treatment (subgroup). Then, a redundant subgroup that is, for example, conditionally independent from the target group given another subgroup, can be suppressed.

For causal analysis, the subgroups are represented by binary variables that are true for an object (case) if it is contained in the subgroup, and false otherwise. For constructing a causal subgroup network, constraint-based methods are particularly suitable because of scalability reasons (cf. [5,10]). These methods make several assumptions (cf. [5]) w.r.t. the data and the correctness of the statistical tests. The crucial condition is the *Markov condition* (cf. [5]) depending on the assumption that the data can be expressed by a Bayesian network: Let X be a node in a causal Bayesian network, and let Y be any node that is not a descendant of X in the causal network. Then, the Markov condition holds if X and Y are independent conditioned on the parents of X.

The CCC and CCU rules [10] described below constrain the possible causal models by applying simple statistical tests: For subgroups s_1, s_2, s_3 represented by binary variables the χ^2-test for independence is utilized for testing their independence $ID(s_1, s_2)$, dependence $D(s_1, s_2)$ and conditional independence $CondID(s_1, s_2|s_3)$, shown below (for the tests user-selectable thresholds are applied, for example, $\mathcal{T}_1 = 1, \mathcal{T}_2 = 3.84$, or higher):

$$ID(s_1, s_2) \longleftrightarrow \chi^2(s_1, s_2) < \mathcal{T}_1 , \qquad D(s_1, s_2) \longleftrightarrow \chi^2(s_1, s_2) > \mathcal{T}_2 ,$$
$$CondID(s_1, s_2|s_3) \longleftrightarrow \chi^2(s_1, s_2|s_3 = 0) + \chi^2(s_1, s_2|s_3 = 1) < 2 \cdot \mathcal{T}_1$$

Thus, the decision of (conditional) (in-)dependence is threshold-based, which is a problem causing potential errors if very many tests are performed. Therefore, we propose a semi-automatic approach featuring interactive analysis of the inferred relations.

Definition 3 (CCC Rule). *Let X, Y, Z denote three variables that are pairwise dependent, that is, $D(X,Y), D(X,Z), D(Y,Z)$; let X and Z become independent when conditioned on Y. In the absence of hidden and confounding variables we may infer that one of the following causal relations exists between X, Y and Z: $X \rightarrow Y \rightarrow Z$, $X \leftarrow Y \rightarrow Z$, $X \leftarrow Y \leftarrow Z$. However, if X has no causes, then the first relation is the only one possible, even in the presence of hidden and confounding variables.*

Definition 4 (CCU Rule). *Let X, Y, Z denote three variables: X and Y are dependent ($D(X,Y)$), Y and Z are dependent ($D(Y,Z)$), X and Z are independent ($ID(X,Z)$), but X and Z become dependent when conditioned on Y ($CondD(X, Z|Y)$). In the absence of hidden and confounding variables, we may infer that X and Z cause Y.*

3 Semi-automatic Causal Subgroup Discovery and Analysis

The proposed approach for causal subgroup discovery features an incremental semi-automatic process. In the following, we first introduce the process model for causal subgroup discovery and analysis. After that, we present the necessary background knowledge for effective causal analysis. Next, we describe a method for constructing an extended causal subgroup network, and discuss how to identify confounded and effect-modified relations.

3.1 Process Model for Causal Subgroup Discovery and Analysis

The steps of the process for semi-automatic causal subgroup discovery are shown in Figure 2:

1. First, the user applies a standard *subgroup discovery* approach, for example, [3,4]. The result of this step is a set of the most interesting subgroups. Optionally, background knowledge contained in the knowledge base can also be applied during subgroup discovery (e.g., as discussed in [11]).
2. Next, we apply *causal analysis* using appropriate background knowledge for a detailed analysis of the discovered associations. Using constraint-based techniques for causal analysis, a (partial) causal subgroup network is constructed.
3. In the *evaluation and validation* phase, the user assesses the (partial) causal network: Since the relations contained in the network can be wrong due to various statistical errors, inspection and validation of the causal relations is essential in order to obtain valid results. Then, the final results are obtained.
4. The user can extend and/or tune the applied background knowledge during the *knowledge extraction* step: Then, the knowledge base can be updated in incremental fashion by including further background knowledge, based upon the discovery results.

3.2 Extended Causal Analysis for Detecting Confounding

Detecting confounding using causal subgroup analysis consists of two main steps that can be iteratively applied:

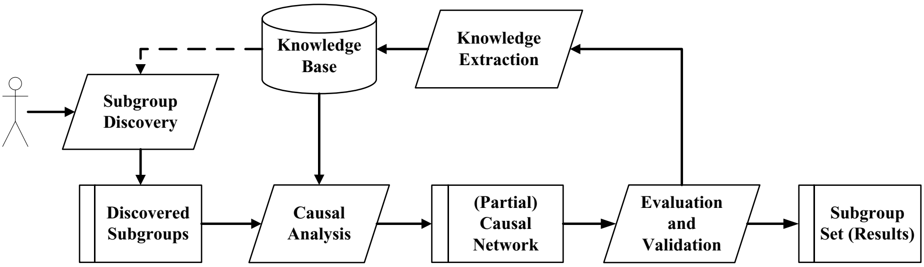


Fig. 2. Process Model for Knowledge-Intensive Causal Subgroup Analysis

1. First, we generate a causal subgroup network considering a target group T, a user-selected set of subgroups U, a set of confirmed (causal) and potentially confounding factors C for any included group, a set of unconfirmed potentially confounding factors P given by subgroups significantly dependent with the target group, and additional background knowledge described below. In addition to causal links, the generated network also contains (undirected) associations between the variables.
2. In the next step, we traverse the network and extract the potential confounded and confounding factors. The causal network and the proposed relations are then presented to the user for subsequent interpretation and analysis. After confounding factors have been confirmed, the background knowledge can then be extended.

In the following we discuss these steps in detail, and also describe the elements of the background knowledge that are utilized for causal analysis.

3.3 Background Knowledge for Causal Analysis

For the causal analysis step we first need to generate an extended causal network capturing the (causal) relations between the subgroups represented by binary variables. In order to effectively generate the network, we need to include background knowledge provided by the user, for example, by a domain specialist. In the medical domain, for example, a lot of background knowledge is already available and can be directly integrated in the analysis phase. Examples include (causal) relations between diseases, and the relations between diseases and their respective findings.
The applicable background consists of two elements:

1. Acausal factors: These include factors represented by subgroups that have no causes; in the medical domain, for example, the subgroup *Age* $\geq$ *70* or the subgroup *Sex = male* have no causes, while the subgroups *BMI = underweight* has certain causes.
2. (Direct) causal relations: Such relations include subgroups that are not only dependent but (directly) causal for the target group/target variable and/or other subgroups. In the medical domain, for example, the subgroup *Body-Mass-Index (BMI)= overweight* is directly causal for the subgroup *Gallstones=probable*.

Depending on the domain, it is often easier to provide acausal information. Direct and indirect causal relations are often also easy to acquire, and can be acquired 'on-the-fly' when applying the presented process. However, in some domains, for example, in the medical domains, it is often difficult to provide non-ambiguous directed relationships between certain variables: One disease can cause another disease and vice versa, under different circumstances. In such cases, the relations should be formalized with respect to both directions, and can still be exploited in the method discussed below. Then, the interpretation performed by the user is crucial in order to obtain valid and ultimately interesting results.

3.4 Constructing an Extended Causal Subgroup Network

Algorithm 1 summarizes how to construct an extended causal subgroup network, based on a technique for basic causal subgroup analysis described in [2,12]. When applying the algorithm, the relations contained in the network can be wrong due to various statistical errors (cf. [5]), especially for the CCU rule (cf. [10]). Therefore, after applying the algorithm, the resulting causal net is presented to the user for interactive analysis.

The first step (lines 1-5) of the algorithm determines for each subgroup pair (including the target group) whether they are independent, based on the inductive principle that the (non in-)dependence of subgroups is necessary for their causality.

In the next step (lines 6-10) we determine for any pair of subgroups whether the first subgroup s_1 is suppressed by a second subgroup s_2, that is, if s_1 is conditionally independent from the target group T given s_2. The χ^2-measure for the target group and s_1 is calculated both for the restriction on s_2 and its complementary subgroup. If the sum of the two test-values is below a threshold, then we can conclude that subgroup s_1 is conditionally independent from the target group. Conditional independence is a sufficient criterion, since the target distribution of s_1 can be explained by the target distribution in s_2, that is, by the intersection. Since very similar subgroups could symmetrically suppress each other, the subgroups are ordered according to their quality. After that, subgroups with a nearly identical extension (but with a lower quality value) can be eliminated.

The next two steps (lines 11-18) check conditional independence between each pair of subgroups given the target group or a third subgroup, respectively. For each pair of conditionally independent groups, the separating (conditioning) group is noted. Then, this separator information is exploited in the next steps, that is, independencies or conditional independencies for pairs of groups derived in the first steps are used to exclude any causal links between the groups. The conditioning steps (lines 6-18) can optionally be iterated in order to condition on combinations of variables (pairs, triples). However, the decisions taken further (in the CCU and CCC rules) may become statistically weaker justified due to smaller counts in the considered contingency tables (e.g., [5,12]).

Direct causal links (line 19) are added based on background knowledge, that is, given subgroup patterns that are directly causal for specific subgroups. In the last step (lines 24-26) we also add conditional associations for dependent subgroups that are not conditionally independent and thus not suppressed by any other subgroups. Such links can later be useful in order to detect the (true) associations considering a confounding factor.

Extending CCC and CCU using Background Knowledge. The CCU and CCC steps (lines 20-23) derive the directions of the causal links between subgroups, based on information derived in the previous steps: In the context of the presented approach, we extend the basic CCC and CCU rules including background knowledge both for the derivation of additional links, and for inhibiting links that contradict the background knowledge. We introduce associations instead of causal directions if these are wrong, or if not enough information for their derivation is available. The rationale behind this principle is given by the intuition that we want to utilize as much information as possible considering the generated causal net.

For the **extended** CCU **rule** we use background knowledge for inhibiting acausal directions, since the CCU rule can be disturbed by confounding and hidden variables.

Algorithm 1. Constructing a causal subgroup net

```
Require: Target group $T$, user-selected set of subgroups $U$, potentially confounding groups $P$,
    background knowledge $B$ containing acausal subgroup information, and known subgroup
    patterns $C \subseteq B$ that are directly causal for other subgroups. Define $\mathcal{S} = U \cup P \cup C$
for all $s_i, s_j \in \mathcal{S} \cup \{T\}, s_i \neq s_j$ do
    if $approxEqual(s_i, s_j)$ then
        Exclude any causalities for the subgroup $s_k$ with smaller correlation to $T$: Remove $s_k$
    if $ID(s_i, s_j)$ then
        Exclude causality: $ex(s_i, s_j) = true$
for all $s_i, s_j \in \mathcal{S}, s_i \neq s_j$ do
    if $\neg ex(s_i, T)$, $\neg ex(s_j, T)$ , or $\neg ex(s_i, s_j)$ then
        if $CondID(s_i, T|s_j)$ then
            Exclude causality: $ex(s_i, T) = true$, and include $s_j$ into $separators(s_i, T)$
            If conditional independencies are symmetric, then select the strongest relation
for all $s_i, s_j \in \mathcal{S}, i < j$ do
    if $\neg ex(s_i, T)$, $\neg ex(s_j, T)$ , or $\neg ex(s_i, s_j)$ then
        if $CondID(s_i, s_j|T)$ then
            Exclude causality: $ex(s_i, s_j) = true$, and include $T$ into $separators(s_i, s_j)$
for all $s_i, s_j, s_k \in \mathcal{S}, i < j, i \neq k, j \neq k$ do
    if $\neg ex(s_i, s_j)$, $\neg ex(s_j, s_k)$ , or $\neg ex(s_i, s_k)$ then
        if $CondID(s_i, s_j|s_k)$ then
            Exclude causality: $ex(s_i, s_j) = true$, and include $s_k$ into $separators(s_i, s_j)$
Integrate direct causal links that are not conditionally excluded considering the sets $C$ and $B$
for all $s_i, s_j, s_k \in \mathcal{S}$ do
    Apply the extended CCU rule, using background knowledge
for all $s_i, s_j, s_k \in \mathcal{S}$ do
    Apply the extended CCC rule, using background knowledge
for all $s_i, s_j, s_k \in \mathcal{S} \cup \{T\}, i < j, i \neq k, j \neq k$ do
    if $\neg CondID(s_i, s_j|s_k)$ then
        Integrate association between dependent $s_i$ and $s_j$ that are not conditionally excluded
```

The causal or associative links do not necessarily indicate direct associations/causal links but can also point to relations enabled by hidden or confounding variables [10].

For the **extended** CCC **rule**, we can use the relations inferred by the extended CCU rule for disambiguating between the causal relations, if the CCU rule is applied in all possible ways: The non-separating condition (conditional dependence) of the relation identified by the CCU rule is not only a sufficient but a necessary condition [10], that is, considering $X \rightarrow Y \leftarrow Z$, with $CondID(X, Z|Y)$. Additionally, we can utilize background knowledge for distinguishing between the causal relations. So, for three variables X, Y, Z with $D(X, Y), D(X, Z), D(Y, Z)$, and $CondID(X, Z|Y)$, if there exists an (inferred) causal link $X \rightarrow Y$ between X and Y, we may identify the relation $X \rightarrow Y \rightarrow Z$ as the true relation. Otherwise, if Y or Z have no causes, then we select the respective relation, for example, $X \leftarrow Y \rightarrow Z$ for an acausal variable Y.

3.5 Identifying Confounded Relations

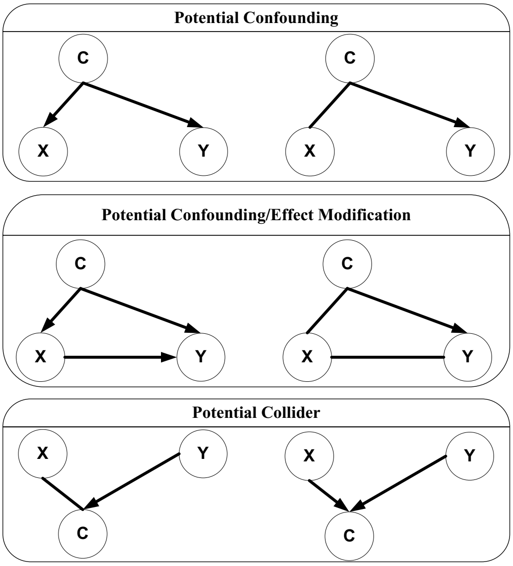


Fig. 3. Examples for Confounded Relations

A popular method for controlling confounding factors is given by *stratification* [6]: For example, in the medical domain a typical confounding factor is the attribute *age*: We can stratify on age groups such as *age* < 30, *age* $30-69$, and *age* ≥ 70. Then, the subgroup – target relations are measured within the different strata, and compared to the (crude) unstratified measure.

It is easy to see, that in the context of the presented approach stratification for a binary variables is equivalent to conditioning on them: If we assess a conditional subgroup – target relation and the subgroup factors become independent (or dependent), then this indicates potential confounding. After constructing a causal net, we can easily identify such relations. Since the causal directions derived by the extended CCC and CCU rules may be ambiguous, user interaction is crucial: In the medical domain, for example, it is often difficult to provide non-ambiguous directed relationships between certain variables: One disease can cause another disease and vice versa, under different circumstances. The network then also provides an intuitive visualization for the analysis.

In order to identify potentially confounded relations and the corresponding variables, as shown in Figure 3, and described below, we just need to traverse the network:

- **Potential Confounding**: If there is an association between two variables X and Y, and the network contains the relations $C \rightarrow X$, $C \rightarrow Y$, and there is no link between X and Y, that is, they are conditionally independent given C, then C is a confounder that inhibits the relation between X and Y. This is also true if there is no causal link between C and X but instead an association.
- **Potential Confounding/Effect Modification**: If the network contains the relations $C \rightarrow X$, $C \rightarrow Y$, and there is also either an association or a causal link between X and Y, then this points to confounding and possible effect modification of the relation between the variables X and Y.
- **Potential Collider (or Confounder)**: If there is no (unconditioned) association between two variables X and Y and the network contains the relations $X \rightarrow C$ and $Y \rightarrow C$, then C is a potential collider: X and Y become dependent by conditioning on C. The variable C is then no confounder in the classical sense, if the (derived) causal relations are indeed true. However, such a relation as inferred by the CCU rule can itself be distorted by confounded and hidden variables. The causal directions could also be inverted, if the association between X and Y is just not strong

enough as estimated by the statistical tests. In this case, C is a potential confounder. Therefore, manual inspection is crucial in order to detect the true causal relation.

4 Examples

We applied a case base containing about 8600 cases taken from the SONOCONSULT system [13] – a medical documentation and consultation system for sonography. The system is in routine use in the DRK-hospital in Berlin/Köpenick, and the collected cases contain detailed descriptions of findings of the examination(s), together with the inferred diagnoses (binary attributes). The experiments were performed using the VIKAMINE system [14] implementing the presented approach.

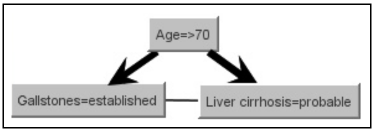


Fig. 4. Confounded Relation: Gallstones and Liver cirrhosis

In the following, we provide some (simplified) examples considering the diagnosis *Gallstones=established* as the target variable. After applying a subgroup discovery method, several subgroups were selected by the user in order to derive a causal subgroup network, and to check the relations w.r.t. possible confounders. These selected subgroups included, for example, the subgroup *Fatty liver=probable or possible* and the subgroup *Liver cirrhosis=probable*.

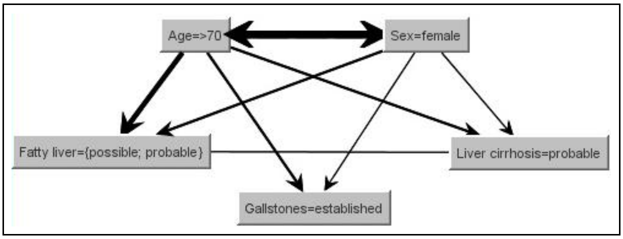


Fig. 5. Confounded Relation: Gallstones, Liver cirrhosis and Fatty Liver

A first result is shown in Figure 4: In this network, the subgroup *Liver cirrhosis=probable* is confounded by the variable *Age*$\geq$*70*. However, there is still an influence on the target variable considering the subgroup *Liver cirrhosis=probable* shown by the association between the subgroups. This first result indicates confounding and effect modification (the strengths of the association between the nodes is also visualized by the widths of the links). A more detailed result is shown in Figure 5: In this network another potential confounder, that is, *Sex=female* is included. Then, it becomes obvious, that both the subgroup *Fatty liver=probable or possible* and the subgroup *Liver cirrhosis=probable* are confounded by the variables *Sex* and *Age*, and the association (shown in Figure 4) between the subgroup *Liver cirrhosis=probable* and the target group is no longer present (In this example the removal of the gall-bladder was not considered which might have an additional effect concerning a medical interpretation).

It is easy to see that the generated causal subgroup network becomes harder to interpret, if many variables are included, and if the number of connections between the nodes increases. Therefore, we provide filters in order to exclude (non-causal) associations. The nodes and the edges of the network can also be color-coded in order to increase their interpretability: Based on the available background knowledge, causal subgroup nodes, and (confirmed) causal directions can be marked. Since the network is first traversed and the potentially confounded relations are reported to the user, the analysis can also be focused towards the respective variables, as a further filtering condition. The user can then analyze selected parts of the network in more detail.

5 Conclusion

In this paper, we have presented a causal subgroup analysis approach for the semi-automatic detection of confounding: Since there is no purely automatic test for confounding [9] the analysis itself depends on background knowledge for establishing an extended causal model/network of the domain. Then, the constructed network can be used to identify potential confounders. In a semi-automatic approach, the network and the potential confounded relations can then be evaluated and validated by the user. We have proposed a comprehensive process model for the presented approach. Furthermore, we have introduced the applicable background knowledge and we have discussed guidelines for its formalization. The benefit and applicability of the presented approach has been shown using examples of a real-world medical system.

In the future, we are planning to integrate an efficient approach for detecting confounding that is directly embedded in the subgroup discovery method with the presented causal methods. So far, we have developed a method for the detection of confounding that is integrated with the subgroup discovery process directly [15]. Combining both approaches, that is, the semi-automatic approach featuring the easily accessible graphical notation in the form of a causal network, and the automatic method for detecting confounding, will potentially increase the effectiveness of the analysis significantly.

Another interesting direction for future work is given by considering further background knowledge for causal analysis, for example, by reusing already formalized knowledge in existing ontologies. Also, the reuse of causal networks that have already been validated by the user and their integration in new evaluations is a promising step for further improvement.

Acknowledgements

This work has been partially supported by the German Research Council (DFG) under grants Pu 129/8-1 and Pu 129/8-2.

References

1. Wrobel, S.: An Algorithm for Multi-Relational Discovery of Subgroups. In: Komorowski, J., Żytkow, J.M. (eds.) PKDD 1997. LNCS, vol. 1263, pp. 78–87. Springer, Heidelberg (1997)
2. Klösgen, W.: 16.3: Subgroup Discovery. In: Handbook of Data Mining and Knowledge Discovery. Oxford University Press, New York (2002)
3. Lavrac, N., Kavsek, B., Flach, P., Todorovski, L.: Subgroup Discovery with CN2-SD. Journal of Machine Learning Research 5, 153–188 (2004)
4. Atzmueller, M., Puppe, F.: SD-Map - A Fast Algorithm for Exhaustive Subgroup Discovery. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS, vol. 4213, pp. 6–17. Springer, Heidelberg (2006)

5. Cooper, G.F.: A Simple Constraint-Based Algorithm for Efficiently Mining Observational Databases for Causal Relationships. Data Min. Knowl. Discov. 1(2), 203–224 (1997)
6. McNamee, R.: Confounding and Confounders. Occup. Environ. Med. 60, 227–234 (2003)
7. Simpson, E.H.: The Interpretation of Interaction in Contingency Tables. Journal of the Royal Statistical Society 18, 238–241 (1951)
8. Fabris, C.C., Freitas, A.A.: Discovering Surprising Patterns by Detecting Occurrences of Simpson's Paradox. In: Research and Development in Intelligent Systems XVI, pp. 148–160. Springer, Heidelberg (1999)
9. Pearl, J.: 6.2 Why There is No Statistical Test For Confounding, Why Many Think There Is, and Why They Are Almost Right. In: Causality: Models, Reasoning and Inference, Cambridge University Press, Cambridge (2000)
10. Silverstein, C., Brin, S., Motwani, R., Ullman, J.D.: Scalable Techniques for Mining Causal Structures. Data Mining and Knowledge Discovery 4(2/3), 163–192 (2000)
11. Atzmueller, M., Puppe, F., Buscher, H.P.: Exploiting Background Knowledge for Knowledge-Intensive Subgroup Discovery. In: Proc. 19th Intl. Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, pp. 647–652 (2005)
12. Kloesgen, W., May, M.: Database Integration of Multirelational Causal Subgroup Mining. Technical report, Fraunhofer Institute AIS, Sankt Augustin, Germany (2002)
13. Huettig, M., Buscher, G., Menzel, T., Scheppach, W., Puppe, F., Buscher, H.P.: A Diagnostic Expert System for Structured Reports, Quality Assessment, and Training of Residents in Sonography. Medizinische Klinik 99(3), 117–122 (2004)
14. Atzmueller, M., Puppe, F.: Semi-Automatic Visual Subgroup Mining using VIKAMINE. Journal of Universal Computer Science 11(11), 1752–1765 (2005)
15. Atzmueller, M., Puppe, F., Buscher, H.-P.: Onto Confounding-Aware Subgroup Discovery. In: Proc. 19th IEEE Intl. Conf. on Tools with Artificial Intelligence (ICTAI 2007), Washington, DC, USA, pp. 163–170. IEEE Computer Society, Los Alamitos (2007)

Using Declarative Specifications of Domain Knowledge for Descriptive Data Mining

Martin Atzmueller and Dietmar Seipel

University of Würzburg
Department of Computer Science
Am Hubland, 97074 Würzburg, Germany
{atzmueller,seipel}@informatik.uni-wuerzburg.de

Abstract. Domain knowledge is a valuable resource for improving the quality of the results of data mining methods. In this paper, we present a methodological approach for providing domain knowledge in a *declarative manner*: We utilize a Prolog knowledge base with facts for the specification of properties of ontological concepts and rules for the derivation of further ad-hoc relations between these concepts. This enhances the *documentation*, *extendability*, and *standardization* of the applied knowledge. Furthermore, the presented approach also provides for potential *automatic verification* and *improved maintenance* options with respect to the used domain knowledge.

1 Introduction

Domain knowledge is a natural resource for knowledge-intensive data mining methods, e.g., [15, 11], and can be exploited for improving the quality of the data mining results significantly. Appropriate domain knowledge can increase the representational expressiveness and also focus the algorithm on the relevant patterns. Furthermore, for increasing the efficiency of the search method, the search space can often be constrained, e.g., [5, 3]. A prerequisite for the successful application and exploitation of domain knowledge is given by a concise description and specification of the domain knowledge. A concise specification also provides for better documentation, extendability, and standardization. Furthermore, maintenance issues and verification of the applied domain knowledge are also enhanced.

In this paper, we present a methodological approach using a declarative specification of domain knowledge. Our context is given by a descriptive data mining method: In contrast to predictive methods, that try to discover a global model for prediction purposes, we focus on methods that provide descriptive patterns for later inspection by the user. These patterns therefore need to be easy to comprehend and to interpret by the user. Thus, besides the objective quality of the patterns as rated by a quality function, such as *support* or *confidence* for association rules, e.g., [1], subjective measures are also rather important: The interestingness of the patterns is also affected by the representational expressiveness of the patterns, and by the focus of the pattern discovery method on the relevant concepts of the domain ontology. In this way, patterns (associations) containing irrelevant and non-interesting concepts can be supressed in order to increase the overall interestingness of the set of the mined patterns.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 149–164, 2009.

Knowledge acquisition is often challenging and costly, a fact that is known as the so-called *knowledge acquisition bottleneck*. Thus, an important idea is to ease the knowledge acquisition by reusing existing domain knowledge, i.e., already formalized knowledge that is contained in existing ontologies or knowledge bases. Furthermore, we aim at simplifying the knowledge acquisition process itself by providing knowledge concepts that are easy to model and to comprehend. We propose high-level knowledge, such as properties of ontological objects for deriving simpler constraint knowledge that can be directly included in the data mining step, as discussed, e.g., in [5, 3]. Modeling such higher-level ontological knowledge, i.e., properties and relations between domain concepts, is often easier for the domain specialist, since it often corresponds to the mental model of the concrete domain.

For specifying the properties and relations of the concepts contained in the domain ontology, we utilize Prolog as a compact and versatile representation; thus, we obtain a suitable representation formalism for domain knowledge. Furthermore, we can automatically derive ad-hoc relations between ontological concepts using rules, and thus also provide a comprehensive overview and summary for the domain specialist. Then, the formalized rules can also be checked with respect to their consistency, i.e., the provided knowledge base and the derivation knowledge can be verified with respect to its semantic integrity.

Altogether, the Prolog rules and facts in the resulting knowledge base automatically serve as a specification and documentation of the domain knowledge that is easy to comprehend, to interpret and to extend by the domain specialist. In addition, new task-specific knowledge can be easily derived by extending and/or modifying the knowledge base. This can be done on several levels depending on the experience of the user: Either new knowledge (facts) can be formalized using the existing types of knowledge, simple rules defining new relations can be introduced, or advanced features can be directly implemented in Prolog. In this way, the declarative features of Prolog allow for a transparent knowledge integration and advancement.

The rest of the paper is organized as follows: We first briefly introduce the context of association rules for descriptive data mining in Section 2. After that, we summarize several types of domain knowledge in Section 3. Next, we describe how the ontological knowledge can be formalized and used for deriving ad-hoc relations, i.e., constraint knowledge in Section 4, and illustrate the approach with practical examples. A real-world case study showing the application of the described types of knowledge is described in Section 5. After that, Section 6 discusses a representation and transformation of the (derived) domain knowledge to XML. Finally, we conclude the paper with a summary in Section 7, and we point out interesting directions for future work.

2 Discovering Association Rules Using Domain Knowledge

In the context of this work, we consider descriptive data mining methods for discovering interesting association rules. Prominent approaches include subgroup discovery methods, e.g., [17, 13, 14, 4], and methods for learning association rules, e.g., [1, 9]. In the following sections we introduce association rules and subgroup patterns and show how the proposed types of domain knowledge are applied for descriptive data mining.

2.1 Association Rules and Subgroup Patterns

Subgroup patterns [12, 5], often provided by conjunctive rules, describe *interesting* properties of subgroups of cases, e.g., *the subgroup of 16-25 year old men that own a sports car are more likely to pay high insurance rates than the people in the reference population*. The main application areas of subgroup discovery [17, 13, 14, 4] are exploration and descriptive induction, to obtain an overview of the relations between a target variable and a set of explaining variables, where variables are attribute/value assignments.

The exemplary subgroup above is then described by the relation between the independent, explaining variables *sex* = *male*, *age* $\leq$ *25*, and *car* = *sports car*, and the dependent, target variable *insurance rate* = *high*. The independent variables are modeled by selection expressions on sets of attribute values. A subgroup pattern is thus described by a subgroup description in relation to a specific target variable. In the context of this work we focus on binary target variables.

Let Ω_A be the set of all attributes. For each attribute $a \in \Omega_A$, a range $dom(a)$ of values is defined. An attribute/value assignment $a = v$, where $v \in dom(a)$, is called a *feature*. A single-relational propositional *subgroup description* is defined as a conjunction

$$sd = e_1 \wedge e_2 \wedge \cdots \wedge e_n$$

of selection expressions $e_i = (a_i \in V_i)$, for subsets $V_i \subseteq dom(a_i)$ of the range of attributes $a_i \in \Omega_A$. An *association rule* [1, 10] is of the form $sd_B \rightarrow sd_H$, where the rule body sd_B and the rule head sd_H are subgroup descriptions. E.g., for the insurance domain, we can consider an association rule showing a combination of potential risk factors for high insurance rates and accidents:

$$\begin{aligned} &\textit{sex} = \textit{male} \wedge \textit{age} \leq \textit{25} \wedge \textit{car} = \textit{sports car} \rightarrow \\ &\qquad\qquad \textit{insurance rate} = \textit{high} \wedge \textit{accident rate} = \textit{high} \end{aligned}$$

Here, the selection expressions are written as features or comparisons specifying intervals. A *subgroup pattern* is a special association rule, namely a horn clause $sd \rightarrow e$, where sd is a subgroup description and e is a feature, called the target variable. For subgroup discovery, usually a fixed target variable is considered.

In general, the quality of an association rule is measured by its support and confidence, and the data mining process searches for association rules with arbitray rule heads and bodies, e.g., using the apriori algorithm [1]. For subgroup patterns there exist various more refined quality measures [13,4]; since an arbitrary quality function can be applied, the anti-monotonic support, which is used in association rule mining, cannot be utilized in the general case.

The applied quality function can also combine the difference of the confidence and the apriori probability of the rule head with the size of the subgroup. Since mining for interesting subgroup patterns is more complicated, usually a fixed, atomic rule head is given as input to the knowledge discovery process.

2.2 Application of Domain Knowledge for Descriptive Data Mining

In the context of this paper, we focus on pattern mining methods for discovering subgroup patterns. The knowledge classes described in Section 3 can be directly integrated in the respective pattern discovery step: While *exclusion* constraints restrict the search space by construction, *aggregation* constraints do not necessarily restrict it, since new values are introduced, but they help to find more understandable results. Also, *combination* constraints inhibit the examination of specified sets of concepts and can prune large, uninteresting areas of the search space. For increasing the representational expressiveness, modifications of the value range of an attribute can be utilized to infer values that are more meaningful for the user. E.g., in the medical domain different aggregated age groups could be considered.

In contrast to existing approaches, e.g., [15, 18] we focus on domain knowledge that can be easily declared in symbolic form. Furthermore, the presented approach features the ability of deriving *simpler* low-level knowledge (constraints) from high-level ontological knowledge. Altogether, the complete knowledge base can be intuitively declared, validated, and inspected by the user, as shown in the following sections.

In general, the search space considered by the data mining methods can be significantly reduced by *shrinking* the value ranges of the attributes. Furthermore, the search can often be focused if only *meaningful* values are taken into account. This usually depends on the considered ontological domain. In the examples below, we consider ordinality and normality information.

Ordinality Information. Let A be an ordinal attribute with an ordered value range

$$dom(A) = \{ v_1, v_2, v_3, v_4, v_5, v_6, v_7 \},$$

where $v_1 < v_2 < \ldots < v_7$. A could be the discretized attribute *body weight* with the following 7 values: *massive underweight*, *strong underweight*, *underweight*, *normal weight*, *overweight*, *strong overweight*, and *massive overweight*.

Ordinality information can be easily applied to derive a restricted domain of aggregated values, where each aggregated value is a meaningful subset of $dom(A)$. The aggregated values could denote different weight groups. Applying ordinality information, we can consider only sub-intervals

$$\langle v_i, v_j \rangle = \{ v_i, v_{i+1}, \ldots, v_j \}$$

of consecutive attribute values, where $v_i \leq v_j$, since subsets with holes, such as $\{v_1,v_3\}$, are irrelevant. Then we obtain only $\binom{7}{2} + 7 = 28$ intervals, instead of all possible $2^7 - 1 = 127$ non-empty subsets:

$$\{v_1\}, \langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \ldots, \langle v_1, v_7 \rangle, \{v_2\}, \langle v_2, v_3 \rangle, \ldots, \langle v_2, v_7 \rangle, \ldots, \langle v_6, v_7 \rangle, \{v_7\}.$$

This already provides for a significant reduction of the search space, if it is applied for many attributes. However, the value range of attributes can be reduced even more using normality/abnormality information as shown below.

Combining Ordinality and Abnormality Information. In the medical domain we often know that a certain attribute value denotes the *normal* value; in our example the normal weight is v_4. This value is often not interesting for the analyst, who might focus on the *abnormal* value combinations. Combining normality and ordinality information, we need to consider only 10 subsets:

$$\begin{aligned} \text{below } v_4 &: \ \{v_1\}, \{v_1, v_2\}, \{v_1, v_2, v_3\}, \{v_2, v_3\}, \{v_3\}, \\ \text{above } v_4 &: \ \{v_5\}, \{v_5, v_6\}, \{v_5, v_6, v_7\}, \{v_6, v_7\}, \{v_7\}. \end{aligned}$$

Depending on the domain, we can reduce the set of values even further. If we are interested only in combinations including the most extreme values v_1 and v_7, which is typical in medicine, then we can further reduce to 6 meaningful subsets:

$$\{v_1\}, \{v_1, v_2\}, \{v_1, v_2, v_3\}, \{v_5, v_6, v_7\}, \{v_6, v_7\}, \{v_7\}.$$

As a variant, we could add one of the intervals $\{v_1, v_2, v_3, v_4\}$ or $\{v_4, v_5, v_6, v_7\}$ with the *normal* value v_4.

For the third case with 6 intervals, the savings of such a reduction of value combinations, which can be derived using ordinality, normality information and interestingness assumptions, are huge: If there are 10 ordinal attributes A with a normal value and with seven values each, then the size of the search space considering all possible combinations is reduced from $127^{10} \approx 10^{21}$ to $6^{10} \approx 6 \cdot 10^8$.

3 Classes and Types of Domain Knowledge

The considered classes of domain knowledge include ontological knowledge and (derived) constraint knowledge, as a subset of the domain knowledge described in [5, 3]. Figure 1 shows the knowledge hierarchy, from the two knowledge classes to the specific types, and the objects they apply to. In the following, we first introduce ontological knowledge. After that, we discuss how constraint knowledge can be derived ad-hoc using ontological knowledge.

3.1 Ontological Knowledge

Ontological knowledge describes general (ad-hoc) properties of the ontological concepts and can be used to infer additional constraints, that can be considered as ad-hoc relations between the domain objects. The applied ontological knowledge, e.g., [5], consists of the following types that can be easily formalized as Prolog ground facts.

Attribute weights denote the relative importance of attributes, and are a common extension for knowledge-based systems [7]. E.g., in the car insurance domain we can state that the attribute *age* is more important than the attribute *car color*, since its assigned weight is higher:

```
weight(age, 4).
weight(car_color, 1).
```

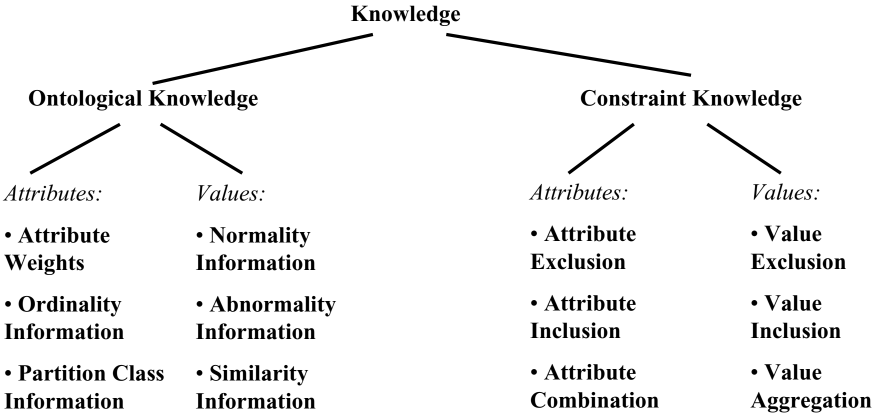


Fig. 1. Hierarchy of (abstract) knowledge classes and specific types

Abnormality/Normality information is usually easy to obtain for diagnostic domains. E.g., in the medical domain the set of *normal* attribute values contains the expected values, and the set of *abnormal* values contains the unexpected ones, where the latter are often more interesting for analysis. Each attribute value is attached with a label specifying a normal or an abnormal state. Normality information only requires a binary label. E.g., for the attribute *temperature* we could have

```
normal(temperature=36.5-38).
```

where the pair `x-y` denotes the right–open interval $[x, y)$. Abnormality information defines several categories. For the range

$$dom(temperature) = \{\, t_1, t_2, t_3, t_4, t_5 \,\},$$

the value t_2 = '$36.5 - 38$' denotes the normal state, while the values t_1 = '< 36.5', t_3 = '$38 - 39$', t_4 = '$39 - 40$', and t_5 = '≥ 40' describe abnormal states. The category 5 indicates the maximum abnormality:

```
abnormality(temperature=0-36.5, 2).
abnormality(temperature=36.5-38, 1).
abnormality(temperature=38-39, 3).
abnormality(temperature=39-40, 5).
abnormality(temperature=40-100, 5).
```

Similarity information between attribute values is often applied in case-based reasoning: It specifies the relative similarity between the individual attribute values. For example, for a nominal attribute *color* with

$$dom(color) = \{\, white, gray, black \,\},$$

we can state that the value *white* is more similar to *gray* than it is to *black*:

```
similarity(color=white, color=gray, 0.5).
similarity(color=white, color=black, 0).
```

Here, the respective similarity value is $s \in [0; 1]$.

Ordinality information specifies if the value range of a nominal attribute can be ordered. E.g., the qualitative attributes *age* and *car size* are ordinal, while *color* is not:

```
ordinal_attribute(age).
ordinal_attribute(car_size).
```

Partition class information provides semantically distinct groups of attributes. These disjoint subsets usually correspond to certain problem areas of the application domain. E.g., in the medical domain such partitions are representing different organ systems like *liver*, *kidney*, *pancreas*, *stomach*, *stomach*, and *intestine*. For each organ system a list of attributes is given:

```
attribute_partition(inner_organs, [
   [fatty_liver, liver_cirrhosis, ...],
   [renal_failure, nephritis, ...], ... ]).
```

3.2 Constraint Knowledge

Constraint knowledge can be applied, e.g., for filtering patterns by their quality and for restricting the search space, as discussed below. We distinguish the following types of constraint knowledge, that can be explicitly provided as ground facts in Prolog. Further basic constraints can be derived automatically as discussed in Section 4.

Exclusion constraints for attributes or features are applied for filtering the domains of attributes and the feature space, respectively. The same applies for *inclusion constraints* for attributes or features, that explicitly state important values and attributes that should be included:

```
dsdk_constraint(exclude(attribute), car_color).
dsdk_constraint(include(attribute), age).
dsdk_constraint(exclude(feature), age=40-50).
```

Value aggregation constraints can be specified in order to form abstracted disjunctions of attribute values, e.g., intervals for ordinal values. For example, for the attribute *age* with

$$dom(age) = \{\, '< 40', '40 - 50', '50 - 70', '\geq 70' \,\}$$

we can use the aggregated values $'< 50'$ (corresponding to the list `[0-40,40-50]`) and $'\geq 50'$ (corresponding to the list `[50-70,70-100]`). In general, aggregated values are not restricted to intervals, but can cover any combination of values.

```
dsdk_constraint(aggregated_values,
   age = [ [0-40,40-50], [50-70,70-100] ]).
```

Attribute combination constraints are applied for filtering/excluding certain combinations of attributes, e.g., if these are already known to the domain specialist.

```
dsdk_constraint(exclude(attribute_pair),
   [car_color, car_size]).
dsdk_constraint(include(attribute_pair),
   [age, car_size]).
```

The data mining process should not compute association rules that contain one of the attributes of an excluded attribute pair in the body and the other attribute in the head.

4 Deriving Constraint Knowledge

Figure 2 shows how ontological knowledge can be used to derive further basic constraints.

Below, we summarize how new constraints can be inferred using ontological knowledge, and show the corresponding Prolog rules for deriving the new constraints. The user can generate a summary of all facts derivable by the provided derivation rules for manual inspection by querying `dsdk_constraint(X,Y)` using backtracking.

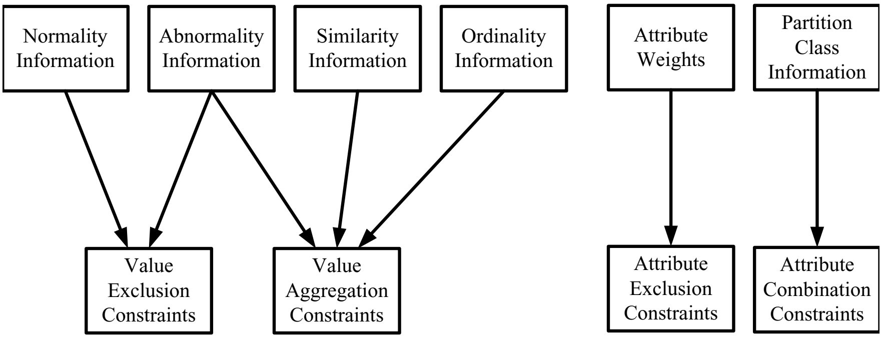

Normality Information
Abnormality Information
Similarity Information
Ordinality Information
Attribute Weights
Partition Class Information
Value Exclusion Constraints
Value Aggregation Constraints
Attribute Exclusion Constraints
Attribute Combination Constraints

Fig. 2. Deriving constraints using ontological knowledge

4.1 Exclusion and Inclusion Constraints

We can construct attribute exclusion constraints using attribute weights to filter the set of relevant attributes by a weight threshold or by subsets of the weight space:

```
dsdk_constraint(exclude(attribute), A) :-
   weight(A, N), N =< 1.
dsdk_constraint(include(attribute), A) :-
   weight(A, N), N > 1.
```

We can apply rules for deriving constraints that can be specified by the user:

```
dsdk_constraint(include(attribute), A) :-
   weight(A, N), N >= 2,
   abnormality(A=_, 5).
```

Using abnormality/normality knowledge we can specify global exclusion constraints for the normal features, i.e. the features whose abnormality is at most 1 (the lowest category 0 denotes the don't care value, while the category 1 explicitly indicates a normal value):

```
dsdk_constraint(exclude(feature), A=V) :-
   abnormality(A=V, N), N =< 1.
```

Partition class information can be used to infer attribute combination constraints in order to exclude pairs of individual attributes that are contained in separate partition classes. Alternatively, inverse constraints can also be derived, e.g., to specifically investigate inter-organ relations in the medical domain.

```
dsdk_constraint(exclude(attribute_pair), [A, B]) :-
   attribute_partition(_, P),
   member(As, P), member(Bs, P), As \= Bs,
   member(A, As), member(B, Bs).
```

Finally, we can use a generic Prolog rule for detecting conflicts w.r.t. these rules and the derived knowledge (*automatic consistency check*):

```
dsdk_constraint(error(include_exclude(X), Y) :-
   dsdk_constraint(include(X), Y),
   dsdk_constraint(exclude(X), Y).
```

4.2 Aggregation Constraints

Aggregation w.r.t. Abnormality. Using similarity or abnormality/normality information we can filter and model the value ranges of attributes. Global abnormality groups can be defined by aggregating values with the same abnormality:

```
dsdk_constraint(aggregated_values, A=Values) :-
   attribute(A),
   findall( Vs,
      abnormality_aggregate(A=Vs),
      Values ).

abnormality_aggregate(A=Vs) :-
   setof( V,
      abnormality(A=V, _),
      Vs ).
```

For a given attribute `A`, the helper predicate `abnormality_aggregate/2` computes the set `Vs` (without duplicates) of all values `V` which have the same abnormality `N` (i.e., the call `abnormality(A=V,N)` succeeds); this grouping is done using the Prolog meta-predicate `setof/3`. For example, for the attribute *temperature* we get 4 groups:

```
?- dsdk_constraint(aggregated_values,
      temperature=Values).
Values = [ [0-36.5], [36.5-38],
   [38-39], [39-40, 40-100] ]
Yes
```

Aggregation w.r.t. Similarity. Also, if the similarity between two attribute values is very high, then they can potentially be analyzed as an aggregated value; this will form a disjunctive selection expression on the value range of the attribute. For example, in the medical domain similar attribute values such as *probable* and *possible* (with different abnormality degrees) can often be aggregated.

The following predicate finds the list `Values` of all sublists `Ws` of the domain `Vs` of an attribute `A`, such that the values in `Ws` are similar in the following sense: if `Ws` = $[w_1, w_2, \ldots, w_n]$, then `similarity`$(A = w_i, A = w_{i+1}, s_i)$ must be given, and it must hold $\mathit{\Pi}_{i=1}^{n-1} s_i \geq 0.5$:

```
dsdk_constraint(aggregated_values, A=Values) :-
   ordinal_attribute(A),
   aggregate_attribute(A),
   domain(A, Vs),
   findall( Ws,
      similar_sub_sequence(A, 0.5, Vs, Ws),
      Values ).
```

Aggregation w.r.t. Normal Value. Ordinality information can be easily used to construct aggregated values, which are often more meaningful for the domain specialist.

- We can consider all adjacent combinations of attribute values, or all ascending/descending combinations starting with the minimum or maximum value, respectively.
- Whenever abnormality information is available, we can partition the value range by the given *normal* value and only start with the most extreme value.

For example, for the ordinal attribute *liver size* with the values

> *1:smaller than normal*, *2:normal*, *3:marginally increased*, *4:slightly increased*, *5:moderately increased*, and *6:highly increased*,

we partition by the *normal* value 2, and we obtain the following aggregated values:

```
[1], [3,4,5,6], [4,5,6], [5,6], [6]
```

Given the domain Vs of an ordinal attribute A, that should be aggregated:

- If the normal value V for A is given, and Vs = $[v_1, \ldots, v_{n-1}, v, v_{n+1}, \ldots, v_m]$, then all starting sequences Ws = $[v_1, v_2, \ldots, v_i]$, where $1 \leq i \leq n-1$, for the sub-domain Vs1 = $[v_1, \ldots, v_{n-1}]$ of all values below V and all ending sequences Ws = $[v_i, \ldots, v_m]$, where $n+1 \leq i \leq m$, for the sub-domain Vs2 = $[v_{n+1}, \ldots, v_m]$ of all values above V are constructed by backtracking.
- If the normal value for A is not given, then all starting (ending) sequences for the full domain Vs are constructed.

```
dsdk_constraint(aggregated_values, A=Values) :-
   ordinal_attribute(A),
   aggregate_attribute(A),
   domain(A, Vs),
   dsdk_split_for_aggregation(A, Vs, Vs1, Vs2),
   findall( Ws,
      ( starting_sequence(Vs1, Ws)
      ; ending_sequence(Vs2, Ws) ),
      Values ).

dsdk_split_for_aggregation(A, Vs, Vs1, Vs2),
   ( normal(A=V),
     append(Vs1, [V|Vs2], Vs)
   ; \+ normal(A=_),
     Vs1 = Vs, Vs2 = Vs ).
```

This type of aggregation constraints using Prolog meta-predicates usually has to be implemented by Prolog experts.

5 Case Study

In this section we describe a case study for the application of the domain knowledge, especially focusing on improving the interestingness of the discoverd results. We use cases taken from the SonoConsult system [8] – a medical documentation and consultation system for sonography. The system is in routine use in the DRK hospital in Berlin/Köpenick and in the University Hospital in Würzburg, and it documents an average of about 300 cases per month in each clinic. The domain ontology of SonoConsult contains about 550 attributes, each having about 5 symbolic values on average, and 221 diagnoses. This indicates the potentially huge search space for subgroup discovery.

For subgroup discovery, we utilized the Vikamine system (*Visual, Interactive and Knowledge-Intensive Analysis and Mining Environment*) for knowledge intensive subgroup mining (`http://vikamine.sourceforge.net`). First, subgroup discovery was performed only using basic attributes and general background knowledge. The users applied attribute weights for feature subset selection. The subgroup discovery algorithm returned many significant subgroups, that supported the validity of the subgroup discovery techniques. However, the results at this stage were not really novel, since they indicated mostly already known dependencies, e.g., the relation between *relative body weight* and *body mass index*.

In the next stage, the expert decided to define new attributes, i.e., *abstracted attributes* which described interesting concepts for analysis. The expert provided 45 derived attributes, that consist of symptom interpretations directly indicating a diagnosis and intermediate concepts which are used in clinical practice, such as *pleura-effusion*, *portal hypertension*, or *pathological gallbladder status*. Furthermore, constraints were formalized, that exclude certain pairs of attributes to restrict the search space. Some examples are depicted in Table 1, where the dependent and independent variables are directly related to each other.

Table 1. Examples of known/uninteresting relations

Target Variable	*Independent Variable*
chronic pancreatitis	*pancreas disease*
pancreas disease	*carcinoma of the pancreas*
body mass index	*relative body weight*

Thus, these pairs were excluded:

```
dsdk_constraint(exclude(attribute_pair), [A, B]) :-
   member([A, B], [
      ['chronic pancreatitis', 'pancreas disease'],
      ['pancreas disease', 'carcinoma of the pancreas'],
      ['body mass index', 'relative body weight'] ]).
```

The impact of the added background knowledge was proven by a greater acceptance of the subgroup discovery results by the expert. However, still too many subgroups were not interesting for the expert, because too many *normal* values were included in the results, e.g., *liver size* = *normal*, or *fatty liver* = *unlikely*. This motivated the application of abnormality information to constrain the value space to the set of *abnormal* values of the attributes. Additionally, the expert suggested to group sets of values into disjunctive value sets defined by abnormality groups, e.g., grouping the values *probable* and *possible* for some attributes. Furthermore, ordinality information was applied to construct grouped values of ordinal attributes like *age* or *liver size*.

As described in Section 4.1, we can infer global exclusion constraints for the normal values and the don't care values:

```
abnormality('liver size'=V, N) :-
   member([V, N], [ [normal, 1], [small, 2],
      [large, 3], [extra_large, 5] ]).
weight('liver size'=3).

?- dsdk_constraint(include(attribute), A).
A = 'liver size'
Yes
?- dsdk_constraint(exclude(feature), A=V).
A = 'liver size', V = normal
Yes
```

The results indicate, that the discussed background knowledge can significantly increase the interestingness of the results, while also pruning the search space, and focusing the discovery process on the interesting patterns. Some examples of clinically interesting subgroup patterns are the following:

$$\begin{aligned}
&\mathit{fatty\ liver} = \mathit{probable} \longrightarrow \mathit{pancreatitis} = \mathit{probable},\\
&\mathit{liver\ size} = \mathit{marginally\ increased} \longrightarrow \mathit{fatty\ liver} = \mathit{probable},\\
&\mathit{aorto\ sclerosis} = \mathit{not\ calcified} \longrightarrow \mathit{fatty\ liver} = \mathit{probable},\\
&\mathit{liver\ size} = \mathit{marginally\ increased} \wedge\\
&\quad \mathit{aorto\ sclerosis} = \mathit{not\ calcified} \longrightarrow \mathit{fatty\ liver} = \mathit{probable}.
\end{aligned}$$

According to the clinicians, the obtained patterns were of significantly higher quality concerning both their interpretability and their interestingness after the domain knowledge had been utilized.

6 XML Representation of the Knowledge

We can also formalize the factual ontological and constraint knowledge in XML – as a standard data representation and exchange format – and transform it to Prolog facts using the XML query and transformation language FNQuery, cf. [16]. FNQuery has been implemented in Prolog and is fully interleaved with the reasoning process of Prolog.

For example, we are using the following XML format for representing constraint knowledge about attributes and classes of attributes:

```
<attribute name="temperature" id="a1" weight="2"
   normal="t2" ordinal="yes" aggregate="yes">
   <domain>
      <value id="t1">&lt;36.5</value>
      <value id="t2">36.5-38</value>
      <value id="t3">38-39</value>
      <value id="t4">&gt;39</value>
   </domain>
   <similarity value="0.2" val1="t3" val2="t4"/>
   ...
</attribute>

<attribute name="fatty_liver" id="io1" ...
<attribute name="liver_cirrhosis" id="io2" ...
<attribute name="renal_failure" id="io3" ...
<attribute name="nephritis" id="io4" ...

<class name="inner_organs">
   <attribute_partition>
      <attributes>
         <attribute id="io1"/>
         <attribute id="io2"/>
      </attributes>
      <attributes>
         <attribute id="io3"/>
         <attribute id="io4"/>
      </attributes>
   </attribute_partition>
</class>
```

The derived Prolog facts can be transformed to XML using FNQuery. Then, using the *Field Notation Grammars* of FNQuery we can generate HTML reports from XML for an intuitive assessment of the (derived) knowledge by the user. Of course, by transforming the XML data we can also provide other specialized output formats, which can, e.g., be used for the integration with other tools.

In addition to such a simple data export of the formalized knowledge, FNQuery also facilitates the functionality for advanced output formats that can then be used for a round-trip engineering of the modeled relations. In this way, an initial knowledge base can either be created using Prolog facts/rules, which are then exported to XML, edited by users using specialized tools, and can be reimported later. This can also be achieved starting with an imported knowledge base, which is then refined using the declarative functionality of Prolog.

7 Conclusions

In this paper, we have presented an approach for the declarative specification of domain knowledge for descriptive data mining. The respective knowledge classes and types can be comprehensively declared, validated, and inspected by the user. The different types of knowledge can be directly integrated in the pattern discovery step of the mining process: using the domain knowledge, the search space can be significantly pruned, uninteresting results can be excluded, and both the representational expressiveness of the ontological objects and the quality of the generated results can be increased.

The structured ontological domain knowledge and the rules for deriving constraint knowledge can be represented nicely in Prolog term structures using function symbols; the factual part of the knowledge can also be represented in XML. For reasoning we make essential use of the following features of Prolog: backtracking, meta-predicates (such as findall, setof, etc.), ease of symbolic computations for term structures. Thus, other declarative languages, such as answer set programming [6], could not be used. Moreover, using the XML query and transformation language FNQuery, we can provide a standard input/output format for high-level domain knowledge, that can be explicitly specified or derived.

We can imagine three kinds of users of the presented approach, depending on their level of experience with the system and/or the presented types of knowledge: Unexperienced users just use the tool as a black box, possibly supported by a (simple) graphical user interface. For these users, the knowledge needs to be pre-formalized by a domain specialist or knowledge engineer. Experienced users can also enter factual knowledge, usually directly as Prolog facts, but also formalized in XML, or supported by specialized (graphical) knowledge acquisition tools. Expert users can program further rules and flexibly extend the system for inferring (specialized) constraints in Prolog. We would assume, that for most applications the biggest portion of the domain knowledge can be formalized by unexperienced or experienced users. Prolog experts are mostly necessary for developing *frameworks*, that can then be filled in by less experienced users.

In the future, we plan to extend the applicable set of domain knowledge by further knowledge elements. Modeling and integrating causal relations [2], for example, is an interesting direction for integrating further important background knowledge. In addition, developing advanced graphical tools supporting inexperienced users is a worthwhile direction for further increasing the overall applicability. Since the factual knowledge can either be represented in XML or in Prolog, appropriate integration, extension, transformation and presentation of the (derived) knowledge is usually easy to accomplish for (external) tools. Another interesting direction for future work is given by an incremental approach for integrating ontological knowledge bases with the obtained data mining results (patterns). The knowledge bases could be extended in a bootstrapping manner, for example, or they could be validated using the discovered patterns.

Acknowledgements

This work has been partially supported by the German Research Council (DFG) under grant Pu 129/8-1 and grant Pu 129/8-2.

References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: Proc. 20th Int. Conf. Very Large Data Bases (VLDB 1994), pp. 487–499. Morgan Kaufmann, San Francisco (1994)
2. Atzmueller, M., Puppe, F.: A Knowledge-Intensive Approach for Semi-Automatic Causal Subgroup Discovery. In: Proc. Workshop on Prior Conceptual Knowledge in Machine Learning and Knowledge Discovery (PriCKL 2007), at the 18th European Conference on Machine Learning (ECML 2007), 11th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2007), Warsaw, Poland, pp. 1–6 (2007)
3. Atzmüller, M., Puppe, F.: A methodological view on knowledge-intensive subgroup discovery. In: Staab, S., Svátek, V. (eds.) EKAW 2006. LNCS, vol. 4248, pp. 318–325. Springer, Heidelberg (2006)
4. Atzmueller, M., Puppe, F.: SD-Map - A Fast Algorithm for Exhaustive Subgroup Discovery. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS, vol. 4213, pp. 6–17. Springer, Heidelberg (2006)
5. Atzmueller, M., Puppe, F., Buscher, H.P.: Exploiting Background Knowledge for Knowledge-Intensive Subgroup Discovery. In: Proc. 19th Intl. Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, pp. 647–652 (2005)
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
7. Baumeister, J., Atzmueller, M., Puppe, F.: Inductive Learning for Case-Based Diagnosis with Multiple Faults. In: Craw, S., Preece, A.D. (eds.) ECCBR 2002. LNCS, vol. 2416, pp. 28–42. Springer, Heidelberg (2002)
8. Huettig, M., Buscher, G., Menzel, T., Scheppach, W., Puppe, F., Buscher, H.P.: A Diagnostic Expert System for Structured Reports, Quality Assessment, and Training of Residents in Sonography. Medizinische Klinik 99(3), 117–122 (2004)
9. Han, J., Pei, J., Yin, Y.: Mining Frequent Patterns Without Candidate Generation. In: Chen, W., Naughton, J., Bernstein, P.A. (eds.) Proc. ACM SIGMOD Intl. Conference on Management of Data (SIGMOD 2000), pp. 1–12. ACM Press, New York (2000)
10. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann, San Francisco (2001)
11. Jaroszewicz, S., Simovici, D.A.: Interestingness of Frequent Itemsets using Bayesian Networks as Background Knowledge. In: Proc. 10th Intl. Conference on Knowledge Discovery and Data Mining (KDD 2004), pp. 178–186. ACM Press, New York (2004)
12. Klösgen, W.: 16.3: Subgroup Discovery. In: Handbook of Data Mining and Knowledge Discovery. Oxford University Press, New York (2002)
13. Klösgen, W.: Explora: A Multipattern and Multistrategy Discovery Assistant. In: Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., Uthurusamy, R. (eds.) Advances in Knowledge Discovery and Data Mining, pp. 249–271. AAAI Press, Menlo Park (1996)
14. Lavrac, N., Kavsek, B., Flach, P., Todorovski, L.: Subgroup Discovery with CN2-SD. Journal of Machine Learning Research 5, 153–188 (2004)
15. Richardson, M., Domingos, P.: Learning with Knowledge from Multiple Experts. In: Proc. 20th Intl. Conference on Machine Learning (ICML 2003), pp. 624–631. AAAI Press, Menlo Park (2003)
16. Seipel, D.: Processing XML-Documents in Prolog. In: Proc. 17th Workshop on Logic Programming (WLP 2002), Dresden (2002)
17. Wrobel, S.: An Algorithm for Multi-Relational Discovery of Subgroups. In: Komorowski, J., Żytkow, J.M. (eds.) PKDD 1997. LNCS, vol. 1263, pp. 78–87. Springer, Heidelberg (1997)
18. Zelezny, F., Lavrac, N., Dzeroski, S.: Using Constraints in Relational Subgroup Discovery. In: Intl. Conference on Methodology and Statistics, Ljubljana, Slovenia, pp. 78–81 (2003)

Integrating Temporal Annotations in a Modular Logic Language

Vitor Nogueira and Salvador Abreu

Universidade de Évora and CENTRIA, Portugal
{vbn,spa}@di.uevora.pt

Abstract. Albeit temporal reasoning and modularity are very prolific fields of research in Logic Programming (LP) we find few examples of their integration. Moreover, in those examples, time and modularity are considered orthogonal to each other. In this paper we propose the addition of temporal annotations to a modular extension of LP such that the usage of a module is influenced by temporal conditions. Besides illustrative examples we also provide an operational semantics together with a compiler, allowing this way for the development of applications based on such language.

1 Introduction

The importance of representing and reasoning about temporal information is well known not only in the database community but also in the artificial intelligence one. In the past decades the volume of temporal data has grown enormously, making modularity a requisite for any language suitable for developing applications for such domains. One expected approach in devising a language with modularity and temporal reasoning is to consider that these characteristics co-exist without any direct relationship (see for instance the language MuTACLP [BMRT02] or [NA06]). Nevertheless we can also conceive a scenario where modularity and time are more integrated, for instance where the usage of a module is influenced by temporal conditions. In this paper we follow the later approach in defining a temporal extensions to a language called Contextual Logic Programming (CxLP) [MP93]. This language is a simple and powerful extension of logic programming with mechanisms for modularity. Recent work not only presented a revised specification of CxLP together with a new implementation for it but also explained how this language could be seen as a shift into the Object-Oriented Programming paradigm [AD03]. Finally, CxLP structure is very suitable for integrating with temporal reasoning since its quite straightforward to add the notion of *time of the context* and let that time help to decide if a certain module is eligible or not to solve a goal.

For temporal representation and reasoning we choose Temporal Annotated Constraint Logic Programming (TACLP) [Frü94,Frü96] since this language supports qualitative and quantitative (metric) temporal reasoning involving both time points and time periods (time intervals) and their duration. Moreover, it allows one to represent definite, indefinite and periodical temporal information.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 165–177, 2009.

The remainder of this article is structured as follows. In Sects. 2 and 3 we briefly overview CxLP and TACLP, respectively. Section 4 presents the temporal extension of CxLP and Sect. 5 relates it with other languages. Conclusions and proposals for future work follows.

2 An Overview of Contextual Logic Programming

For this overview we assume that the reader is familiar with the basic notions of Logic Programming. Contextual Logic Programming (CxLP) [MP93] is a simple yet powerful language that extends logic programming with mechanisms for modularity. In CxLP a finite set of Horn clauses with a given name is designated by *unit*. Using the syntax of GNU Prolog/CX (recent implementation for CxLP [AD03]) consider a unit named `employee` to represent some basic facts about university employees, using `ta` and `ap` as an abbreviation of `t`eaching `a`ssistant and `a`ssociate `p`rofessor, respectively:

```
:-unit(employee(NAME, POSITION)).

item :- employee(NAME, POSITION).
employee(bill, ta).
employee(joe, ap).

name(NAME).
position(POSITION).
```

The main difference between the example above and a plain logic program is the first line that declares the unit name (`employee`) along with the unit arguments (`NAME, POSITION`). Unit arguments help avoid the annoying proliferation of predicate arguments, which occur whenever a global structure needs to be passed around. A unit argument can be interpreted as a "unit global" variable, i.e. one which is shared by all clauses defined in the unit. Therefore, as soon as a unit argument gets instantiated, all the occurrences of that variable in the unit are replaced accordingly.

Suppose also that each employee's position has an associated integer that will be used to calculate the salary. Such relation can be easily expressed by the following unit `index`:

```
:- unit(index(POSITION, INDEX)).

item :-
     index(POSITION, INDEX).

index(ta, 12).
index(ap, 20).

index(INDEX).
position(POSITION).
```

A set of units is designated as a *contextual logic program*. With the units above we can build the program `P = {employee, index}`.

Given that in the same program we can have two or more units with the same name but different arities, to be more precise besides the unit name we should also refer its arity i.e. the number of arguments. Nevertheless, since most of the times there is no ambiguity, we omit the arity of the units. If we consider that `employee` and `index` designate sets of clauses, then the resulting program is given by the union of these sets.

For a given CxLP program, we can impose an order on its units, leading to the notion of *context*. Contexts are implemented as lists of unit designators and each computation has a notion of its *current context*. The program denoted by a particular context is the union of the predicates that are defined in each unit. Moreover, we resort to the *override semantics* to deal with multiple occurrences of a given predicate: only the topmost definition is visible.

To construct contexts, we have the *context extension* operation denoted by :> . The goal U :> G extends the *current context* with unit `U` and resolves goal `G` in the new context. For instance to obtain Bill's position we could do:

```
        | ?- employee(bill, P) :> item

        P = ta
```

In this query we extend the initial empty context `[]` [1] with unit employee obtaining context `[employee(bill, P)]` and then resolve query `item`. This leads to `P` being instantiated with `ta`.

Suppose also that the employee's salary is obtained by multiplying the index of its position by the base salary. To implement this rule consider the unit `salary`:

```
:-unit(salary(SALARY)).

item :-
   position(P),
   [index(P, I)] :< item,
   base_salary(B),
   SALARY is I*B.

base_salary(100).
```

The unit above introduces a new operator denoted by `:<` and called *context switch* such that goal `[index(P, I)] :< item` can be described as invoking `item` in the context `[index(P, I)]`. To better grasp the definition of this unit let us consider another goal:

```
        | ?- employee(bill, P) :> (item, salary(S) :> item).
```

[1] In the GNU Prolog/CX implementation the empty context contains all the standard Prolog predicates such as =/2.

Since we already explained the beginning of this goal, lets see the remaining part. After `salary/1` being added, we are left with the context `[salary(S), employee(bill,ta)]`. The second `item` is evaluated and the first matching definition is found in unit `salary`. Goal `position(P)` is called and since there is no rule for this goal in the current unit (`salary`), a search in the context is performed. Since `employee` is the topmost unit that has a rule for `position(P)`, this goal is resolved in the (reduced) context `[employee(bill, ta)]`. In an informal way, we queried the context for the position of whom we want to calculate the salary, obtaining `ta`. Next, we query the index corresponding to such position, i.e. `[index(ta, I)] :< item`. Finally, to calculate the salary, we just need to multiply the index by the base salary, obtaining as final context `[salary(1200), employee(bill, ta)]` and the answer `S = 1200` and `P = ta`.

3 Temporal Annotated Constraint Logic Programming

This section presents a brief overview of Temporal Annotated Constraint Logic Programming (TACLP) that follows closely Sect. 2 of [RF00]. For a more detailed explanation of TACLP see for instance [Frü96].

We consider the subset of TACLP where time points are totally ordered, sets of time points are convex and non-empty, and only atomic formulae can be annotated. Moreover clauses are free of negation.

Time can be discrete or dense. Time points are totally ordered by the relation $\leq$. We call the set of time points D and suppose that a set of operations (such as the binary operations $+, -$) to manage such points is associated with it. We assume that the time-line is left-bounded by the number 0 and open the future (∞). A *time period* is an interval $[r, s]$ with $0 \leq r \leq s \leq \infty,\ r \in D,\ s \in D$ and represents the convex, non-empty set of time points $\{t \mid r \leq t \leq s\}$. Therefore the interval $[0, \infty]$ denotes the whole time line.

Definition 1 (Annotated Formula). *An annotated formula is of the form $A\alpha$ where A is an atomic formula and α an annotation. Let t be a time point and I be a time period:*

(at) *The annotated formula* **A at t** *means that A holds at time point t.*

(th) *The annotated formula* **A th I** *means that A holds throughout I, i.e. at every time point in the period I.*
A th–annotated formula can be defined in terms of at as: $A\ th\ I \Leftrightarrow\ \forall t\ (t \in I \rightarrow A\ at\ t)$

(in) *The annotated formula* **A in I** *means that A holds at some time point(s) in the time period I, but there is no knowledge when exactly. The in annotation accounts for indefinite temporal information.*
A in–annotated formula can also be defined in terms of at: $A\ in\ I \Leftrightarrow\ \exists t\ (t \in I \wedge A\ at\ t)$.

The set of annotations is endowed with a partial order relation $\sqsubseteq$ which turns into a lattice. Given two annotations α and β, the intuition is that $\alpha \sqsubseteq \beta$ if α is "less informative" than β in the sense that for all formulae A, $A\beta \Rightarrow A\alpha$.

In addition to *Modus Ponens*, TACLP has the following two inference rules:

$$\frac{A\alpha \quad \gamma \sqsubseteq \alpha}{A\gamma} \ \text{rule}\,(\sqsubseteq) \qquad \frac{A\alpha \quad A\beta \quad \gamma = \alpha \sqcup \beta}{A\gamma} \ \text{rule}\,(\sqcup)$$

The rule ($\sqsubseteq$) states that if a formula holds with some annotation, then it also holds with all annotations that are smaller according to the lattice ordering. The rule ($\sqcup$) says that if a formula holds with some annotation and the same formula holds with another annotation then it holds in the least upper bound of the annotations. Assuming $r_1 \leq s_1$, $s_1 \leq s_2$ and $s_2 \leq r_2$, we can summarise the axioms for the lattice operation $\sqsubseteq$ by:

$$in[r_1, r_2] \sqsubseteq in[s_1, s_2] \sqsubseteq in[s_1, s_1] = at\ s_1 = th[s_1, s_1] \sqsubseteq th[s_1, s_2] \sqsubseteq th[r_1, r_2]$$

The axioms of the least upper bound $\sqcup$ can be restricted to [2]:

$$th[s_1, s_2] \sqcup th[r_1, r_2] = th[s_1, r_2] \Leftrightarrow s_1 < r1, r_1 \leq s_2, s_2 < r_2$$

A TACLP *program* is a finite set of TACLP clauses. A TACLP *clause* is a formula of the form $A\alpha \leftarrow C_1, \ldots, C_n, B_1\alpha_1, \ldots, B_m\alpha_m$ $(m, n \geq 0)$ where A is an atom, α and α_i are optional temporal annotations, the C_j's are the constraints and the B_i's are the atomic formulae. Moreover, besides an interpreter for TACLP clauses there is also a compiler that translates them into its CLP form.

4 Temporal Annotations and Contextual Logic Programming

One expected approach in devising a language with modularity and temporal reasoning is to consider that these two characteristics co-exist without any direct relationship. That was the approach followed in [NA06]. Nevertheless we can also conceive a scenario where those concepts are more integrated or more related. In here we take the interaction between modularity and time to that next level by considering that the usage of module is influenced by temporal conditions.

Contextual Logic Programming structure is very suitable for integrating with temporal reasoning since as we will see it is quite straightforward to add the notion of *time of the context* and let that time help deciding if a certain module is eligible or not to solve a goal.

4.1 Language of TCxLP

The basic mechanism of CxLP is called *context search* and can be described as follows: to solve a goal G in a context C, a search is performed until the topmost unit of C that contains clauses for the predicate of G is found. We propose to incorporate temporal reasoning into this mechanism. To accomplish this we add temporal annotations not only to the dynamic part of CxLP (*contexts*) but also to the static one (*units*) and it will be the relation between those two types of annotations that will help to decide if a given unit is eligible to match a goal during a context search.

[2] The least upper bound only has to be computed for overlapping `th` annotations.

Annotating Units. Considering the initial unit definition (i.e. without parameters), adding temporal annotations to these units could be done simply by annotating the unit name. Using GNU Prolog/CX syntax, an annotated unit could be defined as:

```
:- unit(foo) th [1,4].
```

Nevertheless, units with arguments allow for a refinement of the temporal qualification, i.e. we can have several qualifications, one for each possible argument instantiation. As an illustration consider the following example:

Example 1. Temporal annotated unit `bar/1`

```
:- unit(bar(X)).
bar(a) th [1,2].
bar(b) th [3,4].
```

where the annotated facts state that unit `bar` with its argument instantiated to `a` has the annotation `th [1,2]` and with its argument instantiated to `b` has the annotation `th [3,4]`, respectively. Moreover, it should be clear that this way it is still possible to annotate the unit most general descriptor, such as [3]:

```
:- unit(foo(X)).
foo(_) th [1,4].
```

Therefore we propose to temporal annotate the unit descriptors and those annotated facts are designated as the unit *temporal conditions.*

Temporal Annotated Contexts. The addition of time to a context is rather simple and intuitive: instead of a sequence of unit descriptors C we now have a temporally annotated sequence of units descriptors $C\alpha$. This annotation α is called the *time of the context* and by default contexts are implicitly annotated with the current time.

Moreover, remembering that in GNU Prolog/CX context are implemented as lists of units descriptors such as `[u1(X), u2(Y,Z)]`, a temporal annotated context can be for instance `[u1(X), u2(Y,Z)] th [1,4]`.

Relating Temporal Contexts with Temporal Units. Although the relation between temporal annotated contexts and units will be made precise when we present the semantics (sect. 4.2), in this section we illustrate what we mean by "it will be the relation between those two types of annotations (context and units) that will help to decide if a given unit is eligible to match a goal during a context search".

Contexts for a Simple Temporal Unit. Consider the unit `bar/1` of example 1. Roughly speaking, this unit will be eligible to match a goal in a context like `[..., bar(a), ...] in [1,4]` if it is true `bar(a) in [1,4]`.

[3] Please keep in mind that unit `foo/1` is different from the previous unit `foo/0`.

Since one of the unit temporal conditions is `bar(a) th [1,2]` and we know that $in[1,4] \sqsubseteq th[1,2]$, then by inference rule ($\sqsubseteq$) one can derive `bar(a) in [1,4]`. In a similar way we say that this unit its not eligible in the context `[..., bar(b), ...] th [3,6]` (remember that $th[3,6] \not\sqsubseteq th[3,4]$).

Conversely, suppose that unit `bar/1` has some definition for the predicate of an atomic goal `G`, then in order to use such definition in a goal like `[bar(X), ...] in [1,4] :< G` besides the instantiations for the variables of `G`, one also obtains `X = a` or `X = b`.

University Employees. Revisiting the employee example of sect. 2, unit `employee` with temporal information can be written as:

```
:- unit(employee(NAME, POSITION)).

item.

employee(bill, ta) th [2004, inf].
employee(joe, ta) th [2002, 2006].
employee(joe, ap) th [2007, inf].

name(NAME).
position(POSITION).
```

This way it is possible to represent the `history` of the employees positions: `bill` was a teaching assistant (`ta`) between 2002 and 2006. The same person is a associate professor (`ap`) since 2007. Moreover, in this case the rule for predicate `item/0` doesn't need to be `item :- employee(NAME, POSITION)` because the goal `item` is true only if the unit is (temporally) eligible, and for that to happen the unit arguments must be instantiated. To better understand, consider the goal that queries `joe`'s position throughout `[2005,2006]`

```
        | ?-[employee(joe, P)] th [2005,2006] :< item.
```

the evaluation of `item` is true as long as the unit is eligible in the current context, and this is true if `P` is instantiated with `ta` (teaching assistant), therefore we get the answer `P = ta`.

In a similar way we can define a temporal version of unit `index/2` as:

```
:- unit(index(POSITION, INDEX)).

item.

index(ta, 10) th [2000, 2005].
index(ta, 12) th [2006, inf].
index(ap, 19) th [2000, 2005].
index(ap, 20) th [2006, inf].

position(POSITION).
index(INDEX).
```

to express the index raises of each position. Unit `salary` can be defined as:

```
:- unit(salary(SALARY)).
item :-
       position(P),
       index(P, I) :> item,
       base_salary(B),
       SALARY is B*I.

base_salary(100).
```

There is no need to annotate the goals `position(P)` or `index(P, I) :> item` since they are evaluated in a context with the same temporal annotation. To find out `joe`'s salary in 2005 we can do:

```
| ?-[salary(S), employee(joe, P)] at 2005 :< item.
    P = ta
    S = 1000
```

Since `salary` is the topmost unit that defines a rule for `item/0`, the body of the rule for such predicate is evaluated. In order to use the unit `employee(joe, P)` to solve `position(P)`, such unit must satisfy the temporal conditions (`at 2005`), that in this case stands for instantiating `P` with `ta`, therefore we obtain `position(ta)`. A similar reasoning applies for goal `index(ta, I) :> item`, i.e. this `item` is resolved in context `[index(ta, 10), salary(S), employee(joe, ta)] at 2005`. The remainder of the rule body is straightforward, leading to the answer `P = ta` and `S = 1000`.

4.2 Operational Semantics

In this section we assume the following notation: C, C' for contexts, u for unit, $\theta, \sigma, \varphi, \epsilon$ for substitutions, α, β, γ for temporal annotations and $\emptyset, G$ for non-annotated goals.

We also assume a prior computation of the least upper bound for the units `th` annotations. This procedure is rather straightforward and can be describe as: if $A\ th\ I$ and $B\ th\ J$ are in a unit u, such that I and J overlap, then remove those facts from u and insert $A\ th(I \sqcup J)$. This procedure stops when there are no more facts in that conditions. Moreover, the termination is guaranteed because at each step we decrease the size of a finite set, the set of th annotated facts.

Null goal

$$\frac{}{C\alpha \vdash \emptyset[\epsilon]} \tag{1}$$

The null goal is derivable in any temporal annotated context, with the *empty* substitution ϵ as result.

Conjunction of goals

$$\frac{C\alpha \vdash G_1[\theta] \qquad\qquad C\alpha\theta \vdash G_2\theta[\sigma]}{C\alpha \vdash G_1, G_2[\theta\sigma\lceil vars(G_1, G_2)]} \tag{2}$$

To derive the conjunction derive one conjunct first, and then the other in the same context with the given substitutions [4].

Since C may contain variables in unit designators or temporal terms that may be bound by the substitution θ obtained from the derivation of G_1, we have that θ must also be applied to $C\alpha$ in order to obtain the updated context in which to derive $G_2\theta$.

Context inquiry

$$\frac{}{C\alpha \vdash\ :\!>C'\beta[\theta]} \begin{cases} \theta = \mathrm{mgu}(C, C') \\ \beta \sqsubseteq \alpha \end{cases} \qquad (3)$$

In order to make the context switch operation (4) useful, there needs to be an operation which fetches the context. This rule recovers the current context C as a term and unifies it with term C', so that it may be used elsewhere in the program. Moreover, the annotation β must be less (or equal) informative than the annotation α ($\beta \sqsubseteq \alpha$).

Context switch

$$\frac{C'\beta \vdash G[\theta]}{C\alpha \vdash C'\beta\ :\!< G[\theta]} \qquad (4)$$

The purpose of this rule is to allow execution of a goal in an arbitrary temporal annotated context, independently of the current annotated context. This rule causes goal G to be executed in context $C'\beta$.

Reduction

$$\frac{(uC\alpha)\ \theta\sigma \vdash B\theta\sigma[\varphi]}{uC\ \alpha \vdash G[\theta\sigma\varphi\lceil vars(G)]} \begin{cases} H \leftarrow B \in u \\ \theta = \mathrm{mgu}(G, H) \\ (u\theta\sigma)\ \beta \in u \\ \alpha \sqsubseteq \beta \end{cases} \qquad (5)$$

This rule expresses the influence of temporal reasoning on context search. In an informal way we can say that when a goal (G) has a definition ($H \leftarrow B \in u$ and $\theta = \mathrm{mgu}(G, H)$) in the topmost unit ($u$) of the annotated context ($uC\alpha$), and such unit satisfies the temporal conditions, to derive the goal we must call the body of the matching clause, after unification [5]. The verification of the temporal conditions stands for checking if there is a unit temporal annotation ($(u\theta\sigma)\beta \in u$) that is "more informative" than the annotation of the context ($\alpha \sqsubseteq \beta$), i.e. if $(u\theta\sigma)\ \alpha$ is true.

[4] The notation $\delta\lceil V$ stands for the restriction of the substitution δ to the variables in V.

[5] Although this rule might seem complex, that has to do essentially with the abundance of unifications ($\theta\sigma\varphi$)

Context traversal:

$$\frac{C\alpha \vdash G[\theta]}{uC\alpha \vdash G[\theta]} \left\{ pred(G) \notin \overline{u} \right. \tag{6}$$

When none of the previous rules applies and the predicate of `G` isn't defined in the predicates of `u` ($\overline{u}$), remove the top element of the context, i.e. resolve goal G in the supercontext.

Application of the rules. It is almost direct to verify that the inference rules are mutually exclusive, leading to the fact that given a derivation tuple $C\alpha \vdash G[\theta]$ only one rule can be applied.

4.3 Compiler

The compiler for this language can be obtained by combining a program transformation with the compiler for TACLP [Frü96]. Given a unit `u`, such transformation rewrites each predicate `P` in the head of a rule by `P'` and add the following clauses to `u`:

```
P :- Temporal_Conditions, !, P'.
P :- :^ P.
```

stating the resolving `P` is equivalent to invoke `P'`, if the temporal conditions are satisfied. Otherwise `P` must be solved in the supercontext (`:^ P`), i.e. `P` is called in the context obtained from removing `u`.

The temporal condition can be formalised as the conjunction $:< [U|\_]\ \alpha,\ U\alpha$, where the first conjunct queries the context for its temporal annotation (α) and its topmost unit (U), i.e. the current unit. The second conjunct checks if the current unit satisfies the time of the context.

As it should be expected, the compiled language is CxLP with constraints. Finally, since GNU Prolog/CX besides the CxLP primitives also has a constraint solver for finite domains (CLP(FD)), the implementation of this language is direct on such a system.

4.4 Application to Legal Reasoning

Legal reasoning is a very productive field to illustrate the application of these languages. Not only a modular approach is very suitable for reasoning about laws but also time is pervasive in their definition.

The following example was taken from the British Nationality Act and it was presented in [BMRT02] to exemplify the usage of the language MuTACLP. The reason to use an existing example is twofold: not only we consider it to be a simple and concise sample of legal reasoning but also because this way we can give a more thorough comparison with MuTACLP. The textual description of this law can be given as a person `X` obtains the British Nationality at time `T` if:

- X is born in the UK at the time T
- T is after the commencement
- Y is a parent of X
- Y is a British citizen or resident at time T.

Assuming that the temporal unit `person` represents the name and the place where a person was born:

```
:- unit(person(Name, Country)).
person(john, uk) th ['1969-8-10', inf].
```

The temporal annotation of this unit can be interpreted as the person time frame, i.e. when she was born and when she died (if its alive, we represent it by `inf`).

Before presenting the rule for the nationality act we still need to represent some facts about who is a British citizen along with who is parent of whom:

```
   :- unit(british_citizen(Name)).  :- unit(parent(Parent, Son)).

   british_citizen(bob)             parent(bob, john)
        th ['1940-9-6', inf].            th ['1969-8-10', inf].
```

Considering that the commencement date for this law is '1955-1-1', one formalisation of this law in our language is [6]:

```
 th [L, _] :> person(X, uk) :> item, fd_min(L, T),
 '1955-1-1' #=< T,
 at T :> (parent(Y, X) :> item,
                        (british_citizen(Y) :> item;
                         british_resident(Y) :> item)).
```

The explanation of the goal above is quite simple since each line correspond to one condition of the textual description of the law.

5 Related Work

Since [BMRT02] relates MuTACLP with proposals such as Temporal Datalog [OM94] and the work on amalgamating knowledge bases [Sub94], we decided to confine ourselves to the comparison between MuTACLP and our language. MuTACLP (Multi-Theory Temporal Annotated Constraint Logic Programming) is a knowledge representation language that provides facilities for modelling and handling temporal information, together with some basic operators for combining different knowledge bases. Although both MuTACLP and the language here proposed use TACLP (Temporal Annotated Constraint Logic Programming) for handling temporal information, it is in the way that modularity is dealt that they

[6] `fd_min(X, N)` succeeds if N is the minimal value of the current domain of X.

diverge: we follow a dynamic approach (also called *programming-in-the-small*) while MuTACLP engages a static one (also called *programming-in-the-large*).

Moreover, the use of contexts allows for a more compact writing where some of the annotations of the MuTACLP version are subsumed by the annotation of the context. For instance, one of the rules of the MuTACLP version of the example of legal reasoning is:

```
get_citizenship(X) at T <- T >= Jan 1 1955, born(X, uk) at T,
                           parent(Y, X) at T,
                           british_citizen(Y) at T.
```

In [NA06] a similar argument was used when comparing with *relational frameworks* such as the one proposed by Combi and Pozzi in [CP04] for workflow management systems, where relational queries were more verbose that its contextual version.

6 Conclusion and Future Work

In this paper we presented a temporal extension of CxLP where time influences the eligibility of a module to solve a goal. Besides illustrative examples we also provided a compiler, allowing this way for the development of applications based on these languages. Although we provided (in Sect. 4.2) the operational semantics we consider that to obtain a more solid theoretical foundation there is still need for a fixed point or declarative definition.

Besides the domain of application exemplified we are currently applying the language proposed to other areas such as medicine and natural language. Finally, it is our goal to continue previous work [AN06,ADN04] and show that this language can act as the backbone for constructing and maintaining temporal information systems.

References

AD03. Abreu, S., Diaz, D.: Objective: In minimum context. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 128–147. Springer, Heidelberg (2003)

ADN04. Abreu, S., Diaz, D., Nogueira, V.: Organizational information systems design and implementation with contextual constraint logic programming. In: IT Innovation in a Changing World – The 10th International Conference of European University Information Systems, Ljubljana, Slovenia (June 2004)

AN06. Abreu, S., Nogueira, V.: Towards structured contexts and modules. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 436–438. Springer, Heidelberg (2006)

BMRT02. Baldan, P., Mancarella, P., Raffaetà, A., Turini, F.: MuTACLP: A language for temporal reasoning with multiple theories. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS, vol. 2408, pp. 1–40. Springer, Heidelberg (2002)

CP04. Combi, C., Pozzi, G.: Architectures for a temporal workflow management system. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 659–666. Springer, Heidelberg (2004)

Frü94. Frühwirth, T.: Annotated constraint logic programming applied to temporal reasoning. In: Penjam, J. (ed.) PLILP 1994. LNCS, vol. 844, pp. 230–243. Springer, Heidelberg (1994)

Frü96. Frühwirth, T.W.: Temporal annotated constraint logic programming. J. Symb. Comput. 22(5/6), 555–583 (1996)

MP93. Monteiro, L., Porto, A.: A Language for Contextual Logic Programming. In: Apt, K.R., de Bakker, J.W., Rutten, J.J.M.M. (eds.) Logic Programming Languages: Constraints, Functions and Objects, pp. 115–147. MIT Press, Cambridge (1993)

NA06. Nogueira, V., Abreu, S.: Temporal contextual logic programming. In: Fraguas, F.J.L. (ed.) Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP 2006), Madrid, Spain, November 2006. Electronic Notes in Theoretical Computer Science (2006)

OM94. Orgun, M.A., Ma, W.: An overview of temporal and modal logic programming. In: Gabbay, D.M., Ohlbach, H.J. (eds.) ICTL 1994. LNCS, vol. 827, pp. 445–479. Springer, Heidelberg (1994)

RF00. Raffaetà, A., Frühwirth, T.: Semantics for temporal annotated constraint logic programming. In: Raffaetà, A., Frühwirth, T. (eds.) Labelled deduction, pp. 215–243. Kluwer Academic Publishers, Norwell (2000)

Sub94. Subrahmanian, V.S.: Amalgamating knowledge bases. ACM Trans. Database Syst. 19(2), 291–331 (1994)

Visual Generalized Rule Programming Model for Prolog with Hybrid Operators*

Grzegorz J. Nalepa and Igor Wojnicki

Institute of Automatics,
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
gjn@agh.edu.pl, wojnicki@agh.edu.pl

Abstract. The rule-based programming paradigm is omnipresent in a number of engineering domains. However, there are some fundamental semantical differences between it and classic programming approaches. No generic solution for using rules to model business logic in classic software has been provided so far. In this paper a new approach for Generalized Rule-based Programming (GREP) is given. It is based on the use of an advanced rule representation called XTT, which includes an extended attribute-based language, a non-monotonic inference strategy, with an explicit inference control at the rule level. The paper shows, how some typical programming constructs, as well as classic programs can be modelled with this approach. The paper also presents possibilities of an efficient integration of this technique with existing software systems.

It describes the so-called Hybrid Operators in Prolog – a concept which extends the Generalized Rule Based Programming Model (GREP). This extension allows a GREP-based application to communicate with the environment by providing input/output operations, user interaction, and process synchronization. Furthermore, it allows for integration of such an application with contemporary software technologies including Prolog-based code. The proposed Hybrid Operators extend GREP forming a knowledge-based software development concept.

1 Introduction

The rule-based programming paradigm is omnipresent in a number of engineering domains such as control and reactive systems, diagnosis and decision support. Recently, there has been a lot of effort to use *rules* to model business logic in classic software. However, there are some fundamental semantical differences between it, and procedural, or object-oriented programming approaches. This is why no generic modelling solution has been provided so far. The motivation for this paper is to investigate a possibility of modelling some typical programming structures with the rule-based programming with use of on extended forward-chaining rule-based system.

* The paper is supported by the Hekate Project funded from 2007–2009 resources for science as a research project.

In this paper a new approach, the *Generalized Rule Programming* (GREP), is given. It is based on the use of an advanced rule representation, which includes an extended attribute-based language [1], and a non-monotonic inference strategy with an explicit inference control at the rule level. The paper shows, how some typical programming structures (such as loops), as well as classic programs can be modelled using this approach. The paper presents possibilities of efficient integration of this technique with existing software systems in different ways.

In Sect. 2 some basics of rule-based programming are given, and in Sect. 3 some fundamental differences between software and knowledge engineering are identified. Then, in Sect. 4 the extended model for rule-based systems is considered. The applications of this model are discussed in Sect. 5. Some limitations of this approach could be solved by extensions proposed in Sect. 6. The main extension is the concept of *Hybrid Operators* (HOPs), introduced in Sect. 7, which allows for environmental integration of GREP (Sect. 8). Finally, examples of practical HOP applications are presented in Sect. 9. Directions for the future research as well as concluding remarks are given in Sect. 10.

2 Concepts of the Rule-Based Programming

Rule-Based Systems (RBS) constitute a powerful AI tool [2] for knowledge specification in design and implementation of systems in the domains such as: system monitoring and diagnosis, intelligent control, and decision support. For the state-of-the-art in RBS see [3,4,1,5].

In order to design and implement a RBS in an efficient way, the chosen knowledge representation method should support the designer, introducing a scalable *visual representation.* As the number of rules exceeds even relatively very low quantities, it is hard to keep the rule-base consistent, complete, and correct. These problems are related to knowledge-base verification, validation, and testing. To meet security requirements a *formal analysis end verification* of RBS should be carried out [6]. This analysis usually takes place after the design is complete. However, there are design and implementation methods, such as the XTT [7], that allow for on-line verification during the design stage, and gradual refinement of the system.

Supporting rule-base modelling remains an essential aspect of the design process. One of the simplest approaches consists in writing rules in the low-level RBS language, such as one of *Jess* (`www.jessrules.com`). More sophisticated ones are based on the use of some classic visual rule representations i.e. *LPA VisiRule*, (`www.lpa.co.uk`) which uses decision trees. The XTT approach aims at developing a new visual language for *visual rule modelling.*

3 Knowledge in Software Engineering

Rule-based systems (RBS) constitute today one of the most important classes of the so-called Knowledge Based Systems (KBS). RBS have found wide range of industrial applications. However, due to some fundamental differences between knowledge (KE) and software engineering (SE), the knowledge-based approach did not find applications in the mainstream software engineering.

What is important about the KE process, is the fact that it should *capture* the expert knowledge and *represent* it in a way that is suitable for processing (this is the task for a knowledge engineer). The level at which KE should operate is often referred to as *the knowledge level* [8]. In case of KBS there is no single universal engineering approach, or universal modelling method (such as UML in software engineering).

However, software engineering (SE) is a domain where a number of mature and well-proved design methods exist. What makes the SE process different from knowledge engineering is the fact that a systems analyst tries to *model* the *structure* of the real-world information system into the structure of computer software system. So the structure of the software corresponds, to some extent, to the structure of the real-world system, which differs from the KE approach described above.

The fundamental differences between the KE and SE approaches include: declarative vs. procedural point-of-view, semantic gaps present in the SE process, between the requirements, the design, and the implementation, and the application of a gradual abstraction as the main approach to the design. The solution introduced in this paper aims at integrating a classic KE methodology of RBS with SE. It is hoped, that the model considered here, the *Generalized Rule Programming* (GREP), could serve as an effective bridge between SE and KE. The proposed *Hybrid Operators* exension allows for an efficient integration of GREP with existing applications and frameworks.

4 Extended Rule Programming Model

The approach considered in this paper is based on an extended rule-based model. The model uses the *XTT* knowledge method with certain modifications. The XTT method was aimed at forward chaining rule-based systems. In order to be applied to the general programming, it is extended in several aspects.

4.1 XTT – EXtended Tabular Trees

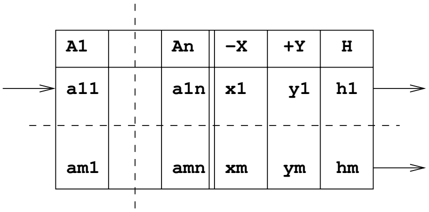
The *XTT* (*EXtended Tabular Trees*) knowledge representation [7], has been proposed in order to solve some common design, analysis and implementation problems present in RBS. In this method three important representation levels has been addressed: *visual* – the model is represented by a hierarchical structure of linked decision tables, *logical* – tables correspond to sequences of decision rules, *implementation* – rules are processed using a Prolog representation.

At the visual level the model is composed of decision tables. A single table is presented in Fig. 1. The table represents a set of rules, having the same attributes. A rule can be read as follows:

$(A11 \in a11) \wedge \ldots (A1n \in a1n) \rightarrow retract(X = x1), assert(Y = y1), do(H = h1)$

In a general case a rule condition can consist of 0 (a rule always fires) to n attributes.

It includes two main extensions compared to the classic RBS: 1) non-atomic attribute values, used both in conditions and decisions, 2) non-monotonic reasoning support, with dynamic assert/retract operations in the decision part. Each table row corresponds to a decision rule. Rows are interpreted top-down. Tables can be linked in a graph-like structure. A link is followed when a rule is fired.



A1		An	-X	+Y	H
a11		a1n	x1	y1	h1
am1		amn	xm	ym	hm

Fig. 1. A single XTT table

At the logical level a table corresponds to a number of rules, processed in a sequence. If a rule is fired and it has a link, the inference engine processes the rule in another table the link points to. The rule is based on an *attributive language* [1]. It corresponds to a *Horn* clause: $\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee h$ where p_i is a literal in SAL (Set Attributive Logic, see [1]) in a form $A_i(o) \in t$ where $o \in O$ is a object referenced in the system, and $A_i \in A$ is a selected attribute of this object (property), $t \subseteq D_i$ is a subset of attribute domain A_i. Rules are interpreted using a unified knowledge and fact base, that can be dynamically modified during the inference process using Prolog-like assert/retract operators in the rule decision part. Rules are implemented using Prolog-based representation, using terms, which is a flexible solution (see [9]). It requires a provided meta-interpreter [10].

This approach has been successfully used to model classic rule-based expert systems. For the needs of general programming described in this paper, some important modifications are proposed.

4.2 Extending XTT into GREP

Considering the use of the XTT method for general applications, there have been several extensions proposed regarding the base XTT model. These are: *Grouped Attributes*, *Attribute-Attribute Comparison*, *Link Labeling*, *Not-Defined Operator*, *Scope Operator*, *Multiple Rule Firing*. Applying these extensions to XTT constitutes *GREP*, that is *Generalized Rule Programming Model*. Additionally there are some examples given in Sect. 5 regarding the proposed extensions.

Grouped Attributes provide means for putting together some number of attributes to express relationships among them and their values. As a result a complex data structure, called a *group*, is created which is similar to constructs present in programming languages (i.e. C structures). A group is expressed as:

$$Group(Attribute1, Attribute2, \ldots, AttributeN)$$

Attributes within a group can be referenced by their name:

$$Group.Attribute1$$

or position within the group:

$$Group/1$$

An application of Grouped Attributes could be expressing spatial coordinates:

$$Position(X, Y)$$

where *Position* is the group name, X and Y are attribute names.

The *Attribute-Attribute Comparison* concept introduces a powerful mechanism to the existing XTT comparison model. In addition to comparing an attribute value against a constant (*Attribute-Value Comparison*) it allows for comparing an attribute value against another attribute value. The *Attribute-Value Comparison* can be expressed as a condition:

```
if (Attribute OPERATOR Value) then ...
```

where `OPERATOR` is a comparison operator i.e. equal, greater then, less than etc., while *Attribute-Attribute Comparison* is expressed as a condition:

```
if (Attribute1 OPERATOR Attribute2) then ...
```

where `OPERATOR` is a comparison operator or a function, in a general case:

```
if (OPERATOR(Attribute1,...,AttributeN)) then ...
```

The operators and functions used here must be well defined.

The *Link Labeling* concept allows to reuse certain XTTs which is similar to subroutines in procedural programming languages. Such a reused XTT can be executed in several contexts. There are incoming and outgoing links. Links might be labeled (see Fig.2). In such a case, if the control comes from a labeled link it has to be directed through an outgoing link with the same label. There can be multiple labeled links for a single rule. If an outgoing link is not labeled it means that if a corresponding rule is fired the link will be followed regardless of the incoming link label. Such a link (or links) might be used to provide a control for exception-like situations.

In the example given in Fig. 2 the outgoing link labeled with $a0$ is followed if the corresponding rule (row) of $reuse0$ is fired and the incoming link is labeled with $a0$. This happens if the control comes from XTT labeled as $table$-$a0$. The outgoing link labeled with $b0$ is followed if the corresponding rule of $reuse0$ is fired and the incoming link is labeled with $b0$; the control comes from $table$-$b0$. If the last rule is fired control is directed as indicated by the last link regardless of the incoming label since the link is not labeled.

The proposed *Not-Defined* (N/D) operator checks if a value for a given attribute has been defined. It has a broad application regarding modelling basic programming structures, i.e. to make a certain rule fired if the XTT is executed

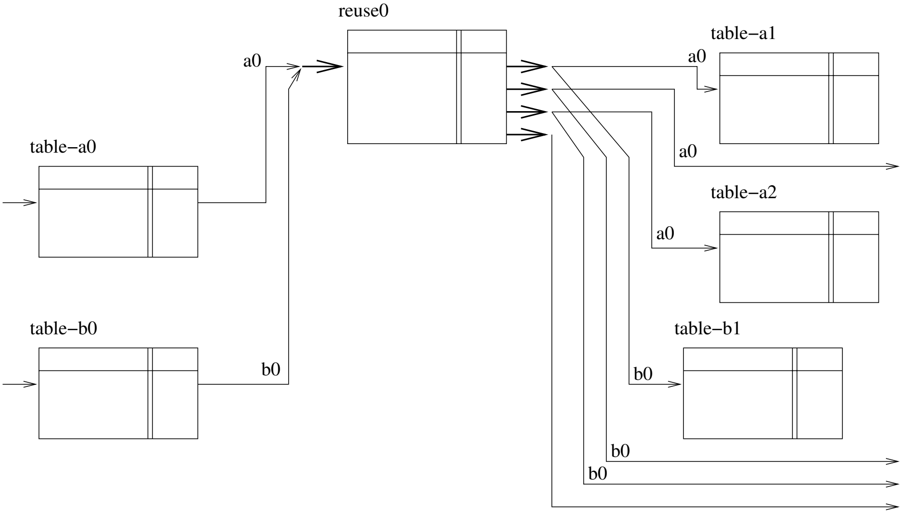


Fig. 2. Reusing XTTs with Link Labeling

for the first time (see Sect. 5.2). It could also be used to model the `NULL` value from the Relational Databases in GREP.

The graphical *Scope Operator* provides a basis for modularized knowledge base design. It allows for treating a set of XTTs as a certain *black box* with well defined input and output (incoming and outgoing links), see Fig. 3. *Scope Operators* can be nested. In such a way a hierarchy of abstraction levels of the system being designed can be built, making modelling of conceptually complex systems easier. The scope area is denoted with a dashed line. Outside the given scope only conditional attributes for the incoming links and the conclusion attributes for the outgoing links are visible. In the given example (Fig. 3) attributes A, B, C are input, while H, I are outputs. Any value changes regarding attributes: E, F, and G are not visible outside the scope area, which consists of *table-b0* and *table=b1* XTTs; no changes regarding values of E, F, and G are visible to *table-a0*, *table-a1* or any other XTT outside the scope.

Since multiple values for a single attribute are allowed, it is worth pointing out that the new inference engine being developed treats them in a more uniform and comprehensive way. If a rule is fired and the conclusion or assert/retract uses a multi-value attribute such a conclusion is executed as many times as there are values of the attribute. It is called *Multiple Rule Firing.* This behavior allows to perform aggregation or set-based operations easily (however, it does not introduce any parallel execution). Some examples are given in Sec. 5.

5 Applications of GREP

This section presents some typical programming constructs developed using the XTT model. It turned out that extending XTT with the modifications described

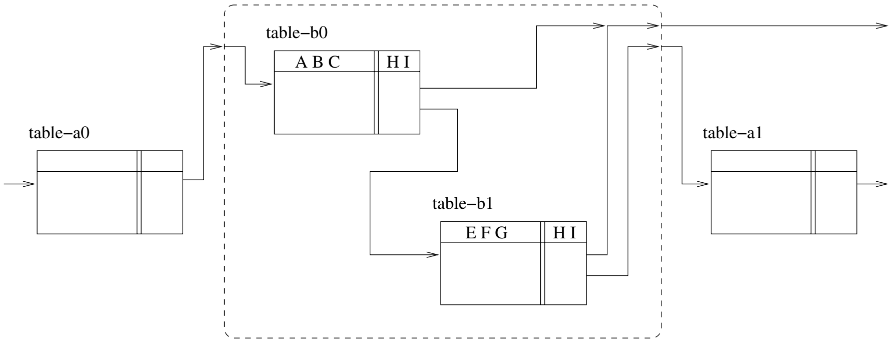


Fig. 3. Introducing Graphical Scope

in Sect. 4.2 allows for applying XTT in other domains than rule-based systems, making it a convenient programming tool.

5.1 Modelling Basic Programming Structures

Two main constructs considered here are: a conditional statement, and a loop.

Programming a conditional statement with rules is both simple and straightforward, since a rule is by definition a conditional statement. In Fig. 4 a single table system is presented. The first row of the table represents the main conditional statement. It is fired if C equals some value v, the result is setting F to $h1$, then the control is passed to other XTT following the outgoing link. The next row implements the `else` statement if the condition is not met ($C \neq v$) then F is set to $h2$ and the control follows the outgoing link. The `F` attribute value is the decision.

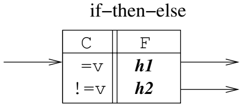
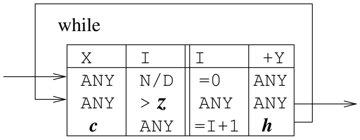


Fig. 4. A conditional statement

A loop can be easily programmed, using the dynamic fact base modification feature (Fig. 5) a simple system implementing the *for*-like loop is presented. Initial execution, as well as the subsequent ones are programmed as a single XTT table. The I attribute serves as the counter. In the body of the loop the value of the decision attribute Y is modified depending on the value of the conditional attribute X. The loop ends when the counter attribute value is greater than the value z. The value ANY in the rule decision indicates that the corresponding attribute value remains unchanged. Using the non-atomic attribute values (an attribute can have a *set* of values) the *foreach* loop could also be constructed.

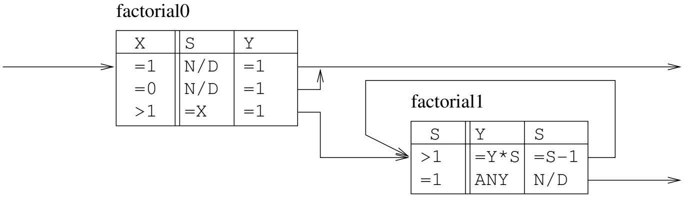


X	I	I	+Y
ANY	N/D	=0	ANY
ANY	> *z*	ANY	ANY
c	ANY	=I+1	***h***

Fig. 5. A loop statement

5.2 Modelling Simple Programming Cases

A set of rules to calculate a factorial is showed in Fig. 6. An argument is given as the attribute X. The calculated result is given as Y. The first XTT ($facactorial0$) calculates the result if $X = 1$ or $X = 0$, otherwise control is passed to $factorial1$ which implements the iterative algorithm using the attribute S as the counter.

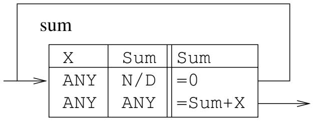


X	S	Y
=1	N/D	=1
=0	N/D	=1
>1	=X	=1

S	Y	S
>1	=Y*S	=S-1
=1	ANY	N/D

Fig. 6. Calculating Factorial of X, result stored as Y

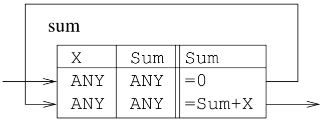
Since an attribute can be assigned more than a single value (i.e. using the assert feature), certain operations can be performed on such a set (it is similar to aggregation operations regarding Relational Databases). An example of sum function is given in Fig. 7. It adds up all values assigned to X and stores the result as a value of Sum attribute. The logic behind it is as follows: If Sum is not defined then make it 0 and loop back. Then, the second rule is fired, since Sum is already set to 0. The conclusion is executed as many times as values are assigned to X. If Sum has a value set by other XTTs prior to the one which calculates the sum, the result is increased by this value. The top arrow could in fact lead to the second row of the table, like in the next example, this would improve the execution.



X	Sum	Sum
ANY	N/D	=0
ANY	ANY	=Sum+X

Fig. 7. Adding up all values of X, result stored as Sum

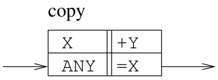
There is also an alternative implementation given in Fig. 8. The difference, comparing with Fig. 7, is that there is an incoming link pointing at the second rule, not the first one. Such an approach utilizes the partial rule execution feature. It means that only the second (and subsequent, if present) rule is investigated. This implementation adds up all values of X regardless if Sum is set in previous XTTs.



X	Sum	Sum
ANY	ANY	=0
ANY	ANY	=Sum+X

Fig. 8. Adding up all values of X, result stored as Sum, an alternative implementation

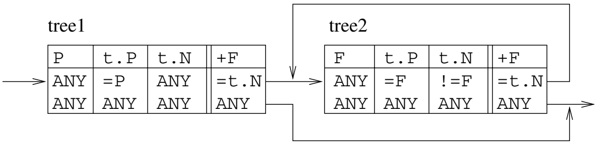
Assigning a set of values to an attribute based on values of another attribute is given in Fig. 9. The given XTT populates Y with all values assigned to X. It uses the XTT assert feature.



X	+Y
ANY	=X

Fig. 9. Copying all values of X into Y

Using the XTT approach, even a complex task such as browsing a tree can be implemented easily. A set of XTTs finding successors of a certain node in a tree is given in Fig. 10. It is assumed that the tree is represented as a group of attributes $t(P, N)$, where N is a node name, and P is a parent node name. The XTTs find all successors of a node which name is given as a value of attribute P (it is allowed to specify multiple values here). A set of successors is calculated as values of F. The first XTT computes immediate child nodes of the given one. If there are some child nodes, control is passed to the XTT labeled *tree2*. It finds child nodes of the children computed by *tree1* and loops over to find children's children until no more child nodes can be found. The result is stored as values of F.



P	t.P	t.N	+F
ANY	=P	ANY	=t.N
ANY	ANY	ANY	ANY

F	t.P	t.N	+F
ANY	=F	!=F	=t.N
ANY	ANY	ANY	ANY

Fig. 10. Finding successors of a node P in a tree, results stored as F

Up to now GREP has been discussed only at the conceptual level, using the visual representation. However, it has to be accompanied by a runtime environment. The main approach considered here is the one of the classic XTT. It is based on using a high-level Prolog representation of GREP. Prolog semantics includes all of the concepts present in GREP. Prolog has the advantages of flexible symbolic representation, as well as advanced meta-programming facilities [10]. The GREP-in-Prolog solution is based on the XTT implementation in Prolog, presented in [9]. In this case a term-based representation is used, with an advanced meta interpreter engine.

6 Motivation for the GREP Extension

In its current stage GREP can successfully model a number of programming constructs and approaches. This proves that GREP can be applied as a general purpose programming environment. However, at the current stage GREP still lacks features needed to replace traditional programming languages.

While proposing extensions to GREP, one should keep in mind, that in order to be both coherent and efficient, GREP cannot make an extensive use of some of the facilities of other languages, even Prolog. Even though Prolog semantics is quite close to the one of GREP there are some important differences – related to the inference (forward in GREP, backward in Prolog) and some programming features such as recursion and unification. On the other hand, GREP offers a clear *visual* representation that supports a high level logic design. The number of failed visual logic programming attempts show, that the powerful semantics of Prolog cannot be easily modelled [11], due to Prolog backward chaining strategy, recursive predicates using multiple clauses, etc. So from this perspective, GREP expressiveness is, and should remain weaker than that of Prolog.

Some shortcomings of GREP are described in the following paragraph. The most important problem regards limited actions/operations in the decision and precondition parts, and input/output operations, including communication with the environment. Actions and operations in the decision and condition parts are limited to using assignment and comparison operators (assert/retract actions are considered assignment operations), only simple predefined operators are allowed. Interaction with the environment of GREP-based application is provided by means of attributes only. Such an interface, while being sufficient for expert systems, becomes insufficient for general purpose software. Particularly there is no basis for executing arbitrary code (external processes, calling a function or method) upon firing a rule. Such an execution would provide an ability to trigger operations outside the GREP application (i.e. running actuators regarding control systems). The only solution possible now in GREP is setting an attribute to a given value. Similarly conditions are not allowed to spawn any additional inference process, i.e. in order to get specific values from arbitrary code (external process, function or method; reading sensor states for control systems). There is also an assumption that setting appropriate attribute triggers some predefined operation, however such operations are not defined by XTTs and they have to be provided by other means. Similarly comparing an attribute with some value might trigger certain operations which leads to value generation for the triggered attribute.

So it can be concluded, that in order to serve as an efficient implementation tool, GREP in its current form should be extended even more. This extension consist in the introduction of *hybrid operators*. Such operators offer processing capabilities for attributes and their values. The *hybrid operators* can serve as generative, restrictive or triggering operations. These are to generate attribute values, restrict admissible attribute values, or trigger arbitrary code execution, respectively.

7 Hybrid Operators

GREP assumes that there is only a single attribute in a column of the XTT table, to be modified by rules. Applying GREP to some cases (see Section 9) indicates a need for more complex operators working on multiple attributes at the same time. To introduce such operators certain changes to the current GREP are necessary. These changes constitute XTTv2 which will be subsequently referred to as XTT in the rest of this paper.

The following extensions, which provide *Hybrid Operators* (HOPs) functionality, are proposed:

- *Defined Operators* (DOT), and
- *Attribute Sets* (ASET).

An operator can be implemented in almost any declarative programming language such as Prolog, procedural or object oriented as: C/C++, Java, or it can even correspond to database queries written in SQL. A Hybrid Operator is an interface between XTT and other programming languages. This is where the name *Hybrid* depicts that such an operator extends XTT with other programming languages and paradigms.

A *Defined Operator* is an operator of the following form:

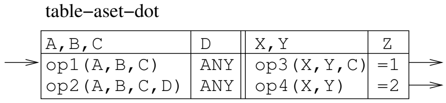
$$Operator(Attribute1, ..., AttributeN)$$

Its functionality is defined by other means and it is not covered by XTT. In general, a rule with a DOT is a regular production rule as any other XTT based one:

```
IF (Operator(Attribute1,...,AttributeN)) THEN ...
```

Since, a DOT is not limited to modify only a single attribute value, it should be indicated which attribute values are to be modified. This indication is provided by ASETs.

An *Attribute Sets* (ASET) is a list of attributes which values are subject to modifications in a given column. The ASET concept is similar to this of *Grouped Attributes* (GA) to some extent. The difference is that a GA defines explicitly a



table–aset–dot

A,B,C	D	X,Y	Z
op1(A,B,C)	ANY	op3(X,Y,C)	=1
op2(A,B,C,D)	ANY	op4(X,Y)	=2

Fig. 11. Hybrid Operators: Attribute Sets (ASET) and Defined Operators (DOT)

relationship among a number of attributes while an ASET provides means for modifying the value of more than one attribute at a time. Attributes within an ASET do not have to be semantically interrelated.

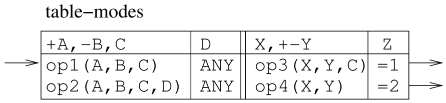
An XTT table with ASET and DOT is presented in Fig. 11. There are two ASETs: attributes A, B, C and X, Y. In addition to the ASETs there are two regular attributes D and Z.

The first column, identified by an ASET: A, B, C, indicates that these three attributes are subject to modification in this column. Similarly a column identified by X, Y indicates that X and Y are subject to modifications in this column. Following the above rules, the operator $op2()$ modifies only A, B, C while D is accessed by it, but it is not allowed to change D values. Similarly operator $op3()$ is allowed to change values of X and Y only.

Depending on where a DOT is placed, either in the condition or conclusion part, values provided by it have different scope. These rules apply to any operators, not only DOTs. If an operator is placed in the condition part, all value modifications are visible in the current XTT table only. If an operator is placed in the conclusion part, all value modifications are applied globally (within the XTT scope, see [12]), and they are visible in other XTT tables.

There are four modes of operation a DOT can be used in: *Restrictive Mode*, *Generative Mode*, *Restrictive-Generative Mode*, and *Setting Mode*. The *Restrictive Mode* narrows number of attribute values and it is indicated as `-`. The *Generative Mode* adds values. It is indicated as `+`. The *Restrictive-Generative Mode* adds or retracts values; indicated as `+-`. Finally, the *Setting Mode* sets attribute values, all previous values are discarded (attributes without `+` or `-` are by default in the *Setting Mode*).

An example with the above modes indicated is given in Fig. 12. An ASET in the first column $(+A, -B, C)$ indicates that A can have some new values asserted, B retracted and C set. An ASET in the third column $(X, + - Y)$ indicates that X has a new set values, while some Y values are retracted and some asserted.



table–modes

+A,-B,C	D	X,+-Y	Z
op1(A,B,C)	ANY	op3(X,Y,C)	=1
op2(A,B,C,D)	ANY	op4(X,Y)	=2

Fig. 12. Hybrid Operators: Modes

Hybrid Operators may be used in different ways. Three main use cases, called schemas are considered, here; these are: *Input*, *Output*, and *Model.* These schemas depict interaction between XTT based application and its environment, i.e. external procedures (written in C, Java, Prolog etc.) accessible through DOTs. This interaction is provided on an attribute basis. The schemas are currently not indicated in XTT tables, such an indication is subject to further research.

The *Input Schema* means that an operator reads data from environment and passes it as attribute values. The *Output Schema* is similar: an operator sends values of a given attribute to the environment. Pure interaction with XTT attributes (no input/output operations) is denoted as the *Model Schema.*

There are several restrictions regarding these schemas. The Input Schema can be used in condition part, while the Output Schema in conclusion part only. Model schemas can be used both in condition or conclusion parts.

Regardless of the schema: Model, Input, or Output, any operation with a DOT involved can be: *blocking* or *non-blocking.* A blocking operation means that the DOT blocks upon accessing an attribute i.e. waiting for a value to be read (Input Schema), write (Output Schema), or set (Model Schema). Such operations may be also *non-blocking*, depending on the functionality needed.

The schemas can provide semaphore-like synchronization based on attributes or event triggered inference i.e. an event unblocking a certain DOT, which spawns a chain of rules to be processed in turn.

There is a drawback regarding Hybrid Operators. It regards validation of an XTT based application. While an XTT model can be formally validated, DOTs cannot be, since they might be created with a non-declarative language (i.e.: C, Java). Some partial validation is feasible if all possible DOT inputs and outputs are known in the design stage. It is a subject to further research.

8 GREP Model Integration with HOP

The XTT model itself is capable of an advanced rule-based processing. However, interactions with the environment were not clearly defined. The Hybrid Operators, introduced here, fill up this gap. An XTT-based logic can be integrated with other components written in Prolog, C or Java. This integration is considered at an architectural level. It follows the Mode-View-Controller (MVC) pattern [13]. In this case the XTT, together with Prolog-based Hybrid Operators, is used to build the application logic: the *model*, where-as other parts of the application are built with procedural or object-oriented languages such C or Java.

The application logic interfaces with object-oriented or procedural components accessible through Hybrid Operators. These components provide means for interaction with an environment which is the user interface and general input-output operations. These components also make it possible to extend the model with arbitrary object-oriented code. There are several scenarios possible regarding interactions between the model and the environment. In general, they can be subdivided into two categories, providing *view* and *controller* functionalities which are output and input respectively.

An input takes place upon checking conditions required to fire a rule. A condition may require input operations. A state of such a condition is determined by data from the environment. Such data could be user input, file contents, a state of an object, a result from a function etc. It is worth pointing out that the input operation could be blocking or non-blocking providing a basis for synchronization with environment. The Input Schema acts as a *controller* regarding the MVC paradigm.

The Output Schema takes place if a conclusion regards an output operation. In such a case, the operation regards general output (i.e. through user interface), spawning a method or function, setting a variable etc. The conclusion also carries its state, which is true or false, depending on whether the output operation succeeded or failed, respectively. If the conclusion fails, the rule fails as well. The Output Schema acts as the *view* regarding MVC.

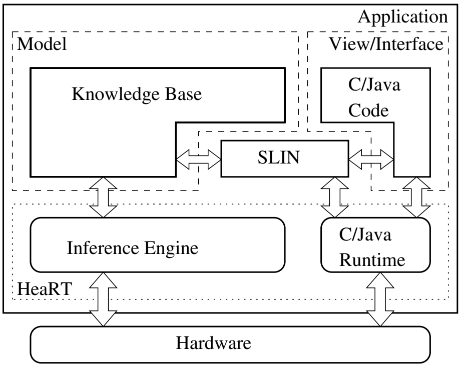


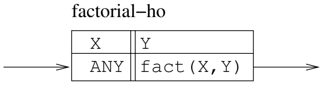
Fig. 13. System Design

There are several components to integrate to provide a working system. They are presented in Fig. 13. The application's logic is given in a declarative way as the Knowledge Base using the XTT representation. Interfaces with other systems (including Human-Computer Interaction) are provided through Hybrid Operators – there is a bridging module between the Knowledge Base and the sequential Code (C/Java language code): the *Sequential Language INterface* (SLIN). It allows communication in both directions. The Knowledge Base can be extended, new facts added by a stimulus coming through SLIN from the View/Interface. There are two types of information passed this way: events generated by the HeaRT (HEkate RunTime) and knowledge generated by the Code (through Hybrid Operators).

9 HOP Applications Examples

Let us consider two generic programming problems examples: factorial calculation and tree browsing.

To calculate the factorial of X and store it as a value of attribute Y XTT tables given in Fig. 6 are needed. It is an iterative approach. Such a calculation can be presented in a simpler form with use of a Hybrid Operator, which provides a factorial function in other programming language, more suitable for this purpose. The XTT table which calculates a factorial of X and stores the result in Y is presented in Fig. 14. In addition to the given XTT table, the $fact()$ operator has to be provided. It can be expressed in Prolog, based on the recursive algorithm, see Fig. 15. Such an approach provides cleaner design at the XTT level.



factorial–ho

X	Y
ANY	fact(X,Y)

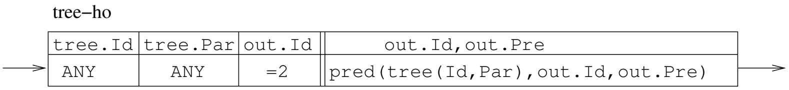
Fig. 14. XTT: factorial function, using a Hybrid Operator

```
fact(0,0).
fact(1,1).
fact(A,B):- A > 1, T is A - 1, fact(T,Z), B is Z * A.
```

Fig. 15. XTT: factorial Hybrid Operator in Prolog

To find all predecessors of a given node, an XTT table is given in Fig. 16. The XTT searches the tree represented by the grouped attribute *tree*/2. Results will be stored in the grouped attribute *out*/2 as *out*(*Id*, *Pre*), where *out.Id* is the node identifier and *out.Pre* is its predecessor node. The search process is narrowed to find predecessors of the node with *out.Id* = 2 only (see attribute *out.Id* in the condition part of the XTT). The search is provided by a hybrid operator *pred*().

The hybrid operator *pred*() is implemented in Prolog (see Fig. 17). It consists of two clauses which implement a recursive tree browsing algorithm. The first argument of the *tree*/3 predicate is to pass the tree structure to browse. Since *tree*/2 in XTT is a grouped attribute it is perceived by Prolog as a structure.



tree–ho

tree.Id	tree.Par	out.Id	out.Id,out.Pre
ANY	ANY	=2	pred(tree(Id,Par),out.Id,out.Pre)

Fig. 16. XTT: Browsing a tree

The hybrid operator *pred*() defined in Fig. 17 works in both directions. It is suitable both for finding predecessors and successors in a tree. The direction depends on which attribute is bound. In the example (Fig. 16) the bound one

```
pred(Tree,Id,Parent):-Tree, Tree=..[_,Id,Parent].
pred(Tree,Id,Parent):-Tree, Tree=..[Pred,Id,X],
                       OtherTree=..[Pred,_,_],
                       pred(OtherTree,X,Parent).
```

Fig. 17. Browsing a tree, a Hybrid Operator in Prolog

is *out.Id*. Bounding *out.Pre* changes the direction, results in search for all successors of the given as *out.Pre* node in the tree. The tree is provided through the first argument of the hybrid operator. The operator can be used by different XTTs, and works on different tree structures.

10 Concluding Remarks

In this paper the results of the research in the field of knowledge and software engineering are presented. The research aims at the unification of knowledge engineering methods with software engineering. The paper presents a new approach for Generalized Rule-based Programming called GREP. It is based on the use of an advanced rule representation called XTT, which includes an attribute-based language, a non-monotonic inference strategy, with explicit inference control at the rule level.

The original contribution of the paper consists in the extension of the XTT rule-based systems knowledge representation method, into GREP, a more general programming solution; as well as the demonstration how some typical programming constructs (such as loops), as well as classic programs (such as factorial, tree search) can be modelled in this approach. The expressiveness and completeness of this solution has been already investigated with respect to a number of programming problems which were showed.

Hybrid Operators extend forward-chaining functionality of GREP with arbitrary inference, including Prolog based backward-chaining. They also provide the basis for an integration with existing software components written in other procedural or object-oriented languages. The examples showed in this paper indicate that GREP and the Hybrid Operators can be successfully applied as a Software Engineering approach. It is worth pointing out that this approach is purely knowledge-based. It constitutes Knowledge-based Software Engineering.

The Generalized Rule Programming concept, extended with Hybrid Operators becomes a powerful concept for software design. It strongly supports the MVC approach: the model is provided by XTT based application extended with Hybrid Operators; both View and Controller functionality (or in other words: communication with the environment, including input/output operations) are provided also through Hybrid Operators. These operators can be implemented in Prolog or (in general case) other procedural or object-oriented language such as Java or C.

Future work will be focused on formulating a complete Prolog representation of GREP extended with HOPs, as well as use cases. Currently a simplified version of GREP has been used to model medium size systems with tens of rules. In these examples the visual scalability of GREP prooved its usefulness. Another important issue concerns the formal verification of the GREP/HOP based systems. It is subject to further research.

References

1. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer, Heidelberg (2006)
2. Negnevitsky, M.: Artificial Intelligence. In: A Guide to Intelligent Systems. Addison-Wesley, Harlow (2002)
3. Liebowitz, J. (ed.): The Handbook of Applied Expert Systems. CRC Press, Boca Raton (1998)
4. Jackson, P.: Introduction to Expert Systems, 3rd edn. Addison Wesley, Reading (1999)
5. Giarratano, J.C., Riley, G.D.: Expert Systems. Thomson (2005)
6. Vermesan, A., Coenen, F. (eds.): Validation and Verification of Knowledge Based Systems. Theory, Tools and Practice. Kluwer Academic Publisher, Boston (1999)
7. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. Systems Science 31, 89–95 (2005)
8. Newell, A.: The knowledge level. Artificial Intelligence 18(1), 87–127 (1982)
9. Nalepa, G.J., Ligęza, A.: Prolog-based analysis of tabular rule-based systems with the "XTT" approach. In: Sutcliffe, G.C.J., Goebel, R.G. (eds.) FLAIRS 2006: proceedings of the nineteenth international Florida Artificial Intelligence Research Society conference, Melbourne Beach, Florida, Artificial Intelligence Research Society, May 11–13, 2006, pp. 426–431. AAAI Press, Menlo Park (2006)
10. Covington, M.A., Nute, D., Vellino, A.: Prolog programming in depth. Prentice-Hall, Englewood Cliffs (1996)
11. Fisher, J.R., Tran, L.: A visual logic. In: Symposium on Applied Computing, pp. 17–21 (1996)
12. Nalepa, G.J., Wojnicki, I.: Visual software modelling with extended rule-based model: a knowledge-based programming solution for general software design. In: Gonzalez-Perez, C., Maciaszek, L.A. (eds.) ENASE 2007: proceedings of the second international conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain, July 23–25, 2007, pp. 41–47. INSTICC Press (2007)
13. Burbeck, S.: Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc). Technical report, Department of Computer Science, University of Illinois, Urbana-Champaign (1992)

The Kiel Curry System KiCS*

Bernd Braßel and Frank Huch

CAU Kiel, Germany
{bbr,fhu}@informatik.uni-kiel.de

Abstract. This paper presents the Kiel Curry System (KiCS), a new implementation of the lazy functional logic language Curry. Its main features beyond other Curry implementations are: flexible search control by means of search trees, referentially transparent encapsulation and sharing across non-determinism.

1 Introduction

The lazy functional logic programming language Curry [14,1] combines the functional and the logical programming paradigms as seamlessly as possible. However, existing implementations of Curry, like PAKCS [13], the Münster Curry Compiler (MCC) [15] and the Sloth system [8], all contain implementation specific integration problems which cannot easily be fixed within these existing systems.

To improve the situation we developed a completely new implementation idea and translation scheme for Curry. The basic idea of the implementation is handling non-determinism (and free variables) by means of tree data structures for the internal representation of the search space. In all existing systems this search space is directly traversed by means of a special search strategy (usually depth-first search). This makes the integration of important features like user-defined search strategies, encapsulation and sharing across non-determinism almost impossible for these systems.

Since the logic computations of Curry are now represented in standard data structures, the resulting programs are more closely related to lazy functional programming (like in Haskell [16]). As a consequence, we decided to use Haskell as target language in our new compiler in contrast to Prolog, as in PAKCS and Sloth, or C in MCC. The advantage of using a lazy functional target language is that most parts of the source language can be left unchanged (especially the deterministic ones) and the efficient machinery behind the Haskell compiler, e.g. the Glasgow Haskell Compiler (GHC), can be reused with a moderate amount of programming effort.

The Kiel Curry System (KiCS) provides an interactive user interface in which arbitrary expressions over the program can be evaluated and a compiler for generating a binary executable for a `main` function of a program, similar to other

* This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 195–205, 2009.

existing systems. The system can be downloaded from `http://www.informatik.uni-kiel.de/prog/mitarbeiter/bernd-brassel/projects/` and requires a functional installation of the GHC. The next sections present the new key features of the Curry implementation KiCS, which improve the seamless integration of functional and logic programming within Curry.

2 Representing Search

To represent search in Haskell, our compiler employs the concept first proposed in [4] and further developed in [7]. There each non-deterministic computation yields a data structure representing the actual search space in form of a tree. The definition of this representation is independent of the search strategy employed and is captured by the following algebraic data type:

```
data SearchTree a = Fail | Value a | Or [SearchTree a]
```

Thus, a non-deterministic computation yields either the successful computation of a completely evaluated term v (i.e., a term without defined functions) represented by `Value` v, an unsuccessful computation (`Fail`), or a branching to several subcomputations represented by `Or` `[`$t_1, \ldots, t_n$`]` where $t_1, \ldots, t_n$ are search trees representing the subcomputations.

The important point is that this structure is provided lazily, i.e., search trees are only evaluated to head normal form. By means of pattern matching on the search tree, a programmer can explore the structure and demand the evaluation of subtrees. Hence, it is possible to define arbitrary search strategies on the structure of search trees. Depth-first search can be defined as follows:

```
depthFirst :: SearchTree a -> [a]
depthFirst (Val v) = [v]
depthFirst Fail    = []
depthFirst (Or ts) = concatMap depthFirst ts
```

Evaluating the search tree lazily, this function evaluates the list of all values in a lazy manner, too. As an example for the definition of another search strategy, breadth-first search can be defined as follows:

```
breadthFirst :: SearchTree a -> [a]
breadthFirst st = unfoldOrs [st]
  where
    partition (Value x) y = let (vs,ors) = y in (x:vs,ors)
    partition (Or xs)   y = let (vs,ors) = y in (vs,xs++ors)
    partition Fail      y = y

    unfoldOrs [] = []
    unfoldOrs (x:xs) =
      let (vals,ors) = foldr partition ([],[]) (x:xs)
       in vals ++ unfoldOrs ors
```

This search strategy is fair with respect to the SearchTree data structure. However, if some computation runs infinitely without generating nodes in the SearchTree, then other solutions within the SearchTree will not be found.

Our concept also allows to formulate a *fair search*, which is also tolerant with respect to non-terminating, non-branching computations. Fair search is realized by employing the multi-threading capabilities of current Haskell implementations. Therefore the definition of fair search is primitive from the point of view of the Curry system.

```
fairSearch :: SearchTree a -> IO [a]
fairSearch external
```

Search trees are obtained by *encapsulated search*. In [4] it is shown in many examples and considerations that the interactions between encapsulation and laziness is very complicated and prone to many problems. [4] also contains a wishlist for future implementations of encapsulation. KiCS is the first implementation to fully meet this wish list. Concretely, there are two methods to obtain search trees:

1. The *current* (top-level) state of search can only be accessed via an io-action `getSearchTree :: a -> IO (SearchTree a)`.
2. A conceptual copy of a given term can be obtained by the primitive operation `searchTree :: a -> SearchTree a`. This copy does not share any of the non-deterministic choices of the main branch of computation, although all deterministic computations are shared. This means that they are only computed once, as described in detail in the next subsection.

These two concepts of encapsulation avoid the conflicts with the other features of functional logic languages which were described in [4].

3 Sharing across Non-determinism

Another key feature for a seamless integration of lazy functional and logic programming is sharing across non-determinism, as the following example shows:

Example 1 (Sharing across Non-Determinism). We consider parser combinators which can elegantly make use of the non-determinism of functional logic languages to implement the different rules of a grammar. A simple set of parser combinators can be defined as follows:

```
type Parser a = String -> (String,a)

success :: a -> Parser a
success r cs = (cs,r)

symb :: Char -> Parser Char
symb c (c':cs) | c==c' = (cs,c)
```

```
(<*>)  :: Parser (a -> b) -> Parser a -> Parser b
p1 <*> p2) str = case p1 str of
                     (str1,f) -> case p2 str1 of
                                   (str2,x) -> (str2,f x)

(<$>) :: (a -> b) -> Parser a -> Parser b
f <$> p = success f <*> p

parse :: Parser a -> String -> a
parse p str = case p str of
                ("",r) -> r
```

As an example for a non-deterministic parser, we construct a parser for the inherently ambiguous, context-free language of palindromes without marked center $L = \{w\overleftarrow{w} \mid w \in \{a,b\}\}$. We restrict to this small alphabet for simplicity of the example. If parsing is possible the parser returns the word w and fails otherwise:

```
pal :: Parser String
pal = ((\ c str _ -> c:str) <$> (symb ’a’ <*> pal <*> symb ’a’))
    ? ((\ c str _ -> c:str) <$> (symb ’b’ <*> pal <*> symb ’b’))
    ? success ""
```

where `? :: a -> a -> a` is the built-in operator for non-deterministic branching, defined as

```
x ? _ = x
_ ? y = y
```

In all Curry implementations the parser `p1` analyses a `String` of length 100 within milliseconds. We call this time t_{parse}.

Unfortunately, this program does not scale well with respect to the time it takes to compute the elements of the list to be parsed. Let’s assume that the time to compute an element within the list `[`$e_1,\dots,e_{100}$`]` is $t \gg t_{parse}$ and constructing the list `[`$e_1,\dots,e_{100}$`]` takes time $100 \cdot t$. Then one would expect the total time to compute `parse pal [`e_1`, . . . ,`e_{100}`]` is $100 \cdot t + t_{parse}$. But measurements for the Curry implementation PAKCS ([13]) show that e.g., for $t = 0.131s$ it takes more than $5000 \cdot t$ to generate a solution for a palindrome `[`$e_1,\dots,e_{100}$`]` and $9910 \cdot t$ to perform the whole search for all solutions. We obtain similar results for all the other existing implementations of lazy functional languages.

The reason is that all these systems do not provide sharing across non-determinism. For each non-deterministic branch in the parser the elements of the remaining list are computed again and again. Only values which are evaluated before non-determinism occurs are shared over the non-deterministic branches. This behavior is not only a matter of leaking implementations within these systems. The development of a formal understanding of sharing across non-determinism has begun recently, only [2,7].

The consequence of the example is that programmers have to avoid using non-determinism, if a function might later be applied to expensive computations. I.e., non-determinism has to be avoided completely. Laziness even complicates this problem: many evaluations are suspended until their value is demanded. A programmer cannot know which values are already computed and, hence, may be used within a non-deterministic computation. Thus, sadly, when looking for possibilities to improve efficiency, the programmer in a lazy functional logic language is well advised to first and foremost try to eliminate the logic features he might have employed.

As an alternative solution, it would also be possible to avoid laziness, e.g., by computing normal forms before performing a non-deterministic computation. But with this solution, the integration of logical features into a lazy functional setting becomes questionable. A strict language, like Oz [17], would allow a smoother combination.

The consequence of both solutions is that the integration of lazy functional and logic programming is not yet as seamless as necessary. Because logic computations are represented by standard data structures in KiCS, the resulting programs are executed in a single global heap. The standard sharing technique updates all computations in this heap. Therefore, achieving sharing across non-determinism does not require any additional effort; it is a natural consequence of the approach.

4 Narrowing Instead of Residuation

Implementations of functional and functional logic languages usually provide numbers as an external data type and reuse the default implementation of the underlying language (e.g. C) for the implementation of operations like `(+)`, `(-)`, `(<=)`, `(==)`. This provides a very efficient implementation of pure computations within the high-level language.

However, in the context of functional logic languages, this approach results in a major drawback: numbers cannot be guessed by means of narrowing. As a consequence, the semantic framework has to be extended, e.g., by *residuation* [9,10]. The idea is that all (externally defined) functions on numbers suspend on free variables as long as their value is unbound. These residuated functions have to be combined with a generator which specifies all possible values of a free variable. As an example, we consider Pythagorean triples (a, b, c) with $a^2 + b^2 = c^2$. This problem can be implemented in the existing Curry implementations by means of the test and generate pattern [3]:

```
pyt | a*a+b*b=:=c*c &
      c =:= member [1..] & b =:= member [1..c] &
      a =:= member [1..c]
    = (a,b,c)
  where a,b,c free

member (y:ys)  = y ? member ys
```

First, the search tests whether a combination of a, b and c is a Pythagorean triple. Since in Curry natural numbers cannot be guessed this test is suspended by means of residuation for external functions (+) and (*). Now the programmer has to add good generator functions which efficiently enumerate the search space.

If, like common in Curry implementations, a depth first search strategy is used, it is especially important to restrict the search space for a and b to a finite domain. In practical applications, finding good generator functions is often much more complicated than in this simple example.

Moreover, residuation sacrifices completeness and detailed knowledge of internals is required to understand why the following very similar definition produces a run-time error (suspension):[1]

```
pyt' | a*a + b*b =:= c*c = generate a b c where a,b,c free

generate a b c | c =:= member [1..] & b =:= member [1..c] &
                 a =:= member [1..c]
  = (a,b,c)
```

On the other hand, the most beautiful standard examples for the expressive power of narrowing are defined for *Peano numbers*:

```
data Peano = O | S Peano

add O     m = m
add (S n) m = S (add m n)

mult O     _ = O
mult (S n) m = add m (mult m n)
```

These functions cannot only be used for their intended purpose. We can also invert them to define new operations. For instance, the subtraction function can be defined by means of add:

```
sub n m | add r m =:= n = r
  where r free
```

Note that we obtain a partial function, since there is no Peano number representation for negative numbers. By means of narrowing, a solution for the constraint add r m =:= n generates a binding for r. This binding is the result of sub.

For a solution of the Pythagorean triples, it is much easier to use Peano numbers instead of predefined integers. We can easily define a solution to this problem as follows:

```
pyt | (a `mult` a) `add` (b `mult` b) =:= (c `mult` c) = (a,b,c)
  where a,b,c free
```

[1] The important difference to the definition of pyt is that the generating constraints like c =:= member [1..] are evaluated *concurrently* with the equation a*a + b*b =:= c*c. In the definition of pyt', in contrast, the implication is that the run-time system should first solve the equation and then satisfy the generating constraints.

It is not necessary to define any generator functions at all. In combination with breadth first search, all solutions are generated.

Furthermore, if the result c of the Pythagorean equation is already known (e.g., $c = 20$) and you are only interested in computing a and b, then it is even possible to compute this information with depth first search. The given equation already restricts the search space to a finite domain.

```
pyt | c =:= intToPeano 20 &
      (a ‘mult‘ a) ‘add‘ (b ‘mult‘ b) =:= (c ‘mult‘ c) = (a,b,c)
  where a,b,c free
```

We obtain the solutions: $(a, b) \in \{(0, 20), (12, 16), (16, 12), (20, 0)\}$.

Unfortunately, using Peano numbers is not appropriate for practical applications, as a simple computation using Peano numbers in PAKCS shows: computing the square of 1000 already takes 7 seconds. As a consequence, the developers of Curry [14] proposed the external type `Int` in combination with external operations which residuate on free variables.

In KiCS numbers are implemented as binary encodings of natural numbers, i.e., numbers greater than 0:

```
data Nat = IHi | O Nat | I Nat
```

The term `IHi` represents a one at the most significant place (the high bit). For example, the term `I (O IHi)` represents the number $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 5$. The representation of each number is unique. If we had a full most significant bit by adding a constructor `OHi`, 5 would also be represented by `I (O (I OHi))` and `I (O (I (O OHi)))`. This would render search spaces on numbers infinite.

In a second step natural numbers are extended with an algebraic sign and a zero to represent arbitrary integers.

```
data Int = Pos Nat | Zero | Neg Nat
```

To avoid redundant representations of numbers by leading zeros, the type `Nat` encodes only positive numbers. The leading one of the binary encoding (its most significant bit), terminates the `Nat` value. Applying the constructor `O` to a `Nat` duplicates it while `I` duplicates and afterwards increments it. Similarly, in pattern matching even numbers can be matched by `O` and odd numbers by `I`.

The implementation of the standard operations for numbers is presented in [6]. To get an impression how the implementation works, we present the definition of the addition for natural numbers here:

```
add :: Nat -> Nat -> Nat
IHi ‘add‘ m   = succ m                1 + m = m + 1
O n ‘add‘ IHi = I n                   2n + 1 = 2n + 1
O n ‘add‘ O m = O (n ‘add‘ m)         2n + 2m = 2 · (n + m)
O n ‘add‘ I m = I (n ‘add‘ m)         2n + (2m + 1) = 2 · (n + m) + 1
I n ‘add‘ IHi = O (succ n)            (2n + 1) + 1 = 2 · (n + 1)
I n ‘add‘ O m = I (n ‘add‘ m)         (2n + 1) + 2m = 2 · (n + m) + 1
I n ‘add‘ I m = O (succ n ‘add‘ y)    (2n + 1) + (2m + 1) = 2(n + 1 + m)
```

where succ is the successor function for Nat. Similarly, all standard definitions for numbers (Nats as well as Ints) can be implemented. In contrast to other Curry implementations, KiCS uses the presented declaration of integers along with the corresponding operations ((+), (*), (-),...) for arbitrary computations on numbers.

For example, the definition of Fibonacci numbers that will be used in the benchmarks of the next section is:

```
fib :: Nat -> Nat
fib n = case n of
  IHi   -> IHi
  O IHi -> IHi
  _     -> let pn=pred n in fib pn +^ fib (pred pn)
```

Here pred is the predecessor function for Nat.

Using these numbers within KiCS, many search problems can be expressed as elegantly as using Peano numbers. For instance, all solutions for the Pythagorean triples problem can in KiCS (in combination with breadth first search) be computed with the following expression:

```
let a,b,c free in a*a + b*b =:= c*c &> (a,b,c)
```

For a fixed c, e.g. previously bound by c =:= 20, all solutions are also computed by depth first search.

The same implementation is used for characters in KiCS, which allows to guess small strings as well. To avoid the overhead related to encoding characters as binary numbers, internally a standard representation as Haskell character is used in the case when characters are not guessed by narrowing.

5 Performance

Although we did not invest much time in optimizing our compiler yet, performance is promising, as the following benchmarks show:

- naive: naive reverse of a list with 10,000 elements
- fib: Fibonacci of 28, PAKCS and MCC use native integers
- fib': Fibonacci of 28, all systems use number representation described in Section 4; for KiCS, fib and fib' are identical
- XML: Read and parse an XML document of almost 2MB
- perm: all possible permutations of the list [1..8] with depth first search (a purely non-deterministic computation)
- permSort: sort the list [12,11..1] by testing permutations (a non-deterministic computation depending heavily on call-time choice)
- narrowLast: compute the last element of a list with 50,000 elements purely by narrowing
- constrLast: compute the last element of a list with 50,000 elements purely by concurrent constraints and unification

- logAppend: a logic version of append, exploiting all advantages of unification, on two lists with 10,000 elements each
- ndParser: non-deterministically parse a string which is very costly to compute. This example depends heavily on sharing across non-determinism as described in Section 1.
- string: time to compute the string parsed in the above example which is generated by using fib' above.

	PAKCS	MCC	KiCS
naive:	16.20 s	1.92 s	1.65 s
fib:	5.80 s	0.10 s	0.10 s
fib':	2.28 s	0.20 s	0.10 s
XML:	-	3.55 s	8.75 s
perm:	0.98 s	0.52 s	0.76 s
permSort:	1.36 s	0.56 s	1.61 s
narrowLast:	1.55 s	0.80 s	1.11 s
constrLast:	1.55 s	0.64 s	1.26 s
logAppend:	0.24 s	0.01 s	0.34 s
ndParser:	241.45 s	17.80 s	0.49 s
string:	15.78 s	1.22 s	0.48 s

Loading the large XML file is not possible in PAKCS. The system crashes, since memory does not suffice. Also, the result that our number representation in PAKCS is faster than the native integers, is quite astonishing.

The comparison of the three systems is not without difficulty. For instance, a good proportion of the differences between MCC and KiCS on one hand and PAKCS on the other hand is due to memory allocation. Since both of the former systems use finally the gcc compiler, there are good possibilities to determine, e.g., the heap size allowed. We have made use of this possibility, allowing both MCC and KiCS the space needed for optimal performance, but we have found no way to do the same for PAKCS. Therefore, our benchmarks measure mostly the performance of the SICStus garbage collector.

The benchmarks also reveal that both unification and concurrent constraints in KiCS have to be improved. We think that the overhead of over 30 times in comparison to the MCC in example logAppend is real and has already come up in practical applications and also in comparison to PAKCS.

The last benchmark demonstrates what we have described in Section 1. Both, MCC and PAKCS have to compute the elements of the parsed string again and again. As the string is of length 30, this accumulates to an average overhead of 15 times of what is needed to compute the string.

6 Concluding Remarks

There exist many libraries to support real world applications written in Curry in the PAKCS system, e.g., for high level database programming [5], web applications [12] and graphical user interfaces [11]. We have already ported most of the external functions provided in PAKCS and, thus, all applications can also be compiled with KiCS.

The implementation of KiCS has just started and there should be a lot of room for optimizations, which we want to investigate for future work. However, we think our approach is promising and it has already proven to be a good platform to experiment with new features for lazy functional-logic languages to further improve the integration of the two programming paradigms.

References

1. Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G.: Operational semantics for declarative multi-paradigm languages. Journal of Symbolic Computation 40(1), 795–829 (2005)
2. Antoy, S., Brown, D.W., Chiang, S.-H.: On the correctness of bubbling. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 35–49. Springer, Heidelberg (2006)
3. Antoy, S., Hanus, M.: Functional logic design patterns. In: Hu, Z., Rodríguez-Artalejo, M. (eds.) FLOPS 2002. LNCS, vol. 2441, pp. 67–87. Springer, Heidelberg (2002)
4. Braßel, B., Hanus, M., Huch, F.: Encapsulating non-determinism in functional logic computations. Journal of Functional and Logic Programming 2004(6) (2004)
5. Braßel, B., Hanus, M., Müller, M.: High-level database programming in curry. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 316–332. Springer, Heidelberg (2008)
6. Braßel, B., Huch, F., Fischer, S.: Declaring numbers. In: Proceedings of the 16th International Workshop on Funcional and (Constraint) Logic Programming (WFLP 2007), Paris, France (July 2007)
7. Braßel, B., Huch, F.: On a tighter integration of functional and logic programming. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 122–138. Springer, Heidelberg (2007)
8. Gallego, E.J., Marino, J.: An overview of the Sloth 2005 Curry system. In: Hanus, M. (ed.) First Workshop on Curry and Functional Logic Programming. ACM Press, New York (2005)
9. Hanus, M.: On the completeness of residuation. In: Proc. of the 1992 Joint International Conference and Symposium on Logic Programming, pp. 192–206. MIT Press, Cambridge (1992)
10. Hanus, M.: The integration of functions into logic programming: From theory to practice. Journal of Logic Programming 19&20, 583–628 (1994)
11. Hanus, M.: A functional logic programming approach to graphical user interfaces. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, pp. 47–62. Springer, Heidelberg (2000)
12. Hanus, M.: Type-oriented construction of web user interfaces. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2006), pp. 27–38. ACM Press, New York (2006)
13. Hanus, M., Antoy, S., Braßel, B., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R., Steiner, F.: PAKCS: The Portland Aachen Kiel Curry System (2006), `http://www.informatik.uni-kiel.de/~pakcs/`
14. Hanus, M. (ed.): Curry: An integrated functional logic language, vers. 0.8.2 (2006), `http://www.informatik.uni-kiel.de/~curry`

15. Lux, W., Kuchen, H.: An efficient abstract machine for curry. In: Beiersdörfer, K., Engels, G., Schäfer, W. (eds.) Informatik 1999 — Annual meeting of the German Computer Science Society (GI), pp. 390–399. Springer, Heidelberg (1999)
16. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, Cambridge (2003)
17. Smolka, G.: The Oz programming model. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 324–343. Springer, Heidelberg (1995)

Narrowing for First Order Functional Logic Programs with Call-Time Choice Semantics[*]

Francisco J. López-Fraguas, Juan Rodríguez-Hortalá,
and Jaime Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. In a recent work we have proposed *let*-rewriting, a simple one-step relation close to ordinary term rewriting but able, via local bindings, to express sharing of computed values. In this way, *let*-rewriting reflects the call-time choice semantics for non-determinism adopted by modern functional logic languages, where programs are rewrite systems possibly non-confluent and non-terminating. In this paper we extend that work providing a notion of *let*-narrowing which is adequate for call-time choice as proved by soundness and completeness results of *let*-narrowing with respect to *let*-rewriting. Completeness is based on a lifting lemma for *let*-rewriting similar to Hullot's lifting lemma for ordinary rewriting and narrowing. Our work copes with first order, left linear, constructor-based rewrite systems with no other restrictions about confluence, termination or presence of extra variables in right-hand sides of rules.

1 Introduction

Programs in functional-logic languages (see [11] for a recent survey) are constructor-based term rewriting systems possibly non-confluent and non-terminating, as happens in the following example:

$$\begin{array}{ll} coin \to 0 & repeat(X) \;\to\; X\!:\!repeat(X) \\ coin \to 1 & heads(X\!:\!Y\,:\!Ys) \;\to\; (X,Y) \end{array}$$

Here *coin* is a 0-ary non-deterministic function that can be evaluated to 0 or 1, and *repeat* introduces non-termination on the system. The presence of non-determinism enforces to make a decision about *call-time* (also called *singular)* or *run-time choice* (or *plural*) semantics [14,21]. Consider for instance the expression $heads(repeat(coin))$:

- run-time choice gives $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$ as possible results. Rewriting can be used for it as in the following derivation:

$$\begin{array}{l} heads(repeat(coin)) \to heads(coin : coin : \ldots) \to \\ (coin, coin) \to (0, coin) \to (0,1) \end{array}$$

[*] This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03) and Promesas-CAM (S-0505/TIC/0407).

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 206–222, 2009.

- under call-time choice we obtain only the values $(0, 0)$ and $(1, 1)$ (*coin* is evaluated only once and the corresponding value must be *shared*).

Modern functional-logic languages like Curry [12] or Toy [17] adopt call-time choice, as it seems more appropriate in practice. Although the classical theory of TRS (term rewriting systems) is the basis of some influential papers in the field, specially those related to *needed narrowing* [2], it cannot really serve as technical foundation if call-time choice is taken into account, because ordinary rewriting corresponds to run-time choice, and therefore is unsound (i.e., produces undesired values) for call-time choice, as shown in the example above. This was a main motivation for the *CRWL*[1] framework [8,9], that is considered an adequate semantic foundation (see [10]) for the paradigm.

From an intuitive point of view there must be a strong connection between *CRWL* and classical rewriting, but this has not received much attention in the past. Recently, in [15] we have started to investigate such a connection, by means of *let-rewriting*, that enhances ordinary rewriting with explicit *let*-bindings to express sharing, in a similar way to what was done in [18] for the λ-calculus. A discussion of the reasons for having proposed *let*-rewriting instead of using other existing formalisms like term graph-rewriting [19,6] or specific operational semantics for FLP [1] can be found in [15].

In this paper we extend *let*-rewriting to a notion of *let-narrowing*. Our main result will be a *lifting lemma* for *let*-narrowing in the style of Hullot's one for classical narrowing [13].

We do not pretend that *let*-narrowing as will be presented here can replace advantageously existing versions of narrowing like needed narrowing [2] or natural narrowing [7], which are well established as appropriate operational procedures for functional logic programs. *Let*-narrowing is more a complementary proposal: needed or natural narrowing express refined strategies with desirable optimality properties to be preserved in practice, but they need to be patched in implementations in order to achieve sharing (otherwise they are unsound for call-time choice). *Let*-rewriting and *let*-narrowing intend to be the theoretical basis of that patch, so that sharing needs not be left anymore out of the scope of technical works dealing with rewriting-based operational aspects of functional logic languages. Things are then well prepared for recasting in the future the needed or natural strategies to the framework of *let*-rewriting and narrowing.

The rest of the paper is organized as follows. Section 2 contains a short presentation of *let*-rewriting. Section 3 tackles our main goal, extending *let*-rewriting to *let*-narrowing and proving its soundness and completeness. Finally, Section 4 outlines some conclusion and future lines of research.

2 Preliminaries

2.1 Constructor-Based Term Rewrite Systems

We consider a first order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint sets of *constructor* and defined *function* symbols respectively, all them

[1] *CRWL* stands for *C*onstructor-based *Re**W*riting *L*ogic.

with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects.

The set *Exp* of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like *Exp*, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that *Exp* stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$.

We will frequently use *one-hole contexts*, defined as:

$$Cntxt \ni \mathcal{C} ::= [\ ] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n) \quad (h \in CS^n \cup FS^n)$$

The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively by:

$$[\ ][e] = e \qquad h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$$

The set *Subst* of *substitutions* consists of finite mappings $\theta : \mathcal{V} \longrightarrow Exp$ (i.e., mappings such that $\theta(X) \neq X$ only for finitely many $X \in \mathcal{V}$), which extend naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e, and $\theta\theta'$ for the composition of substitutions, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $ran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. If $dom(\theta) \cap dom(\theta') = \emptyset$, then $\theta \uplus \theta'$ is defined by $X(\theta \uplus \theta') = X\theta$ ($X\theta'$ resp.) for $X \in dom(\theta)$ ($X \in dom(\theta')$ resp.). Given $W \subseteq \mathcal{V}$ we write by $\theta|_W$ the restriction of θ to W, and $\theta|_{\setminus D}$ is a shortcut for $\theta|_{(\mathcal{V} \setminus D)}$. We will sometimes write $\theta = \sigma[W]$ instead of $\theta|_W = \sigma|_W$. In most cases we will use c-substitutions $\theta \in CSubst$, verifying that $X\theta \in CTerm$ for all $X \in dom(\theta)$. We say that e *subsumes* e', and write $e \preceq e'$, if $e\theta = e'$ for some θ; we write $\theta \preceq \theta'$ if $X\theta \preceq X\theta'$ for all variables X and $\theta \preceq \theta'[W]$ if $X\theta \preceq X\theta'$ for all $X \in W$.

The *shell* $|e|$ of an expression e, that represents the outer constructed part of e, is defined as follows:

$$\begin{aligned} |X| &= X \\ |c(e_1, \ldots, e_n)| &= c(|e_1|, \ldots, |e_n|) \\ |f(e_1, \ldots, e_n)| &= \bot \end{aligned}$$

Here $\bot$ represents the semantic value *undefined*; when $\bot$ is added to CS as a new constant, we obtain the sets $Expr_\bot$ and $CTerm_\bot$ of partial expressions and c-terms respectively, playing an important role in the *CRWL* semantics [8,9], but of secondary importance here since $\bot$ will not appear in programs nor computations. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$.

A *constructor-based term rewriting system* $\mathcal{P}$ (also called *program* along this paper) is a set of c-rewrite rules of the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\overline{t}$. Notice that we allow e to contain *extra variables*, i.e., variables not occurring in $\overline{t}$. Given a program $\mathcal{P}$, its associated rewrite relation $\to_{\mathcal{P}}$ is defined as: $\mathcal{C}[l\theta] \to_{\mathcal{P}} \mathcal{C}[r\theta]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\theta \in Subst$. Notice that in the definition of $\to_{\mathcal{P}}$ it is allowed for θ to instantiate extra variables to any expression. We write $\overset{*}{\to}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\to_{\mathcal{P}}$. In the following, we will usually omit the reference to $\mathcal{P}$.

2.2 Rewriting with Local Bindings

In [15] we have proposed *let-rewriting* as an alternative rewriting relation for constructor-based term rewriting sytems that uses *let*-bindings to get an explicit formulation of sharing, i.e., call-time choice semantics. Although the primary goal for this relation was to establish a closer relationship between classical rewriting and the *CRWL*-framework of [9], *let*-rewriting is interesting in its own as a simple one-step reduction mechanism for call-time choice. This relation manipulates *let-expressions*, defined as:

$$LExp \ni e ::= X \mid h(e_1, \ldots, e_n) \mid let\ X = e_1\ in\ e_2$$

where $X \in \mathcal{V}$, $h \in CS \cup FS$, and $e_1, \ldots, e_n \in LExp$. The notation $let\ \overline{X = e}\ in\ e'$ abbreviates $let\ X_1 = e_1\ in\ \ldots\ in\ let\ X_n = e_n\ in\ e'$. The notion of one-hole context is also extended to the new syntax:

$$\mathcal{C} ::= [\,] \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \mid h(\ldots, \mathcal{C}, \ldots)$$

The *shell* of a *let*-expression is defined by adding the case $|let\ X = e_1\ in\ e_2| = |e_2|[X/|e_1|]$ to the definition of the previous section. Notice that shells do not contain *let*s.

Free and *bound* (or *produced*) variables of $e \in LExp$ are defined as:

$$\begin{array}{l} FV(X) = \{X\};\ FV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} FV(e_i); \\ FV(let\ X = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \backslash \{X\}); \\ BV(X) = \emptyset;\ BV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} BV(e_i); \\ BV(let\ X = e_1\ in\ e_2) = BV(e_1) \cup BV(e_2) \cup \{X\} \end{array}$$

Notice that the given definition of $FV(let\ X = e_1\ in\ e_2)$ precludes recursive *let*-bindings in the language since the possible occurrences of X in e_1 are not considered bound and therefore refer to a 'different' X. As frequent with formalisms that bind variables, we assume implicitly that an expression can be identified with the one resulting of consistently renaming its bound variables. We assume also a variable convention according to which the same variable symbol does not occur free and bound within an expression. Moreover, to keep simple the management of substitutions, we assume that whenever θ is applied to an expression $e \in LExp$, the necessary renamings are done in e to ensure that $BV(e) \cap (dom(\theta) \cup ran(\theta)) = \emptyset$. With all these conditions the rules defining application of substitutions are simple while avoiding variable capture:

$$X\theta = \theta(X); \quad h(\overline{e})\theta = h(\overline{e\theta}); \quad (let\ X = e_1\ in\ e_2)\theta \ = \ (let\ X = e_1\theta\ in\ e_2\theta)$$

The *let*-rewriting relation $\to_l$, as defined in [15], is shown in Figure 1. The rule **(Fapp)** performs a rewriting step in a proper sense, using a rule of the program and a *matching substitution* θ. Note that only c-substitutions are allowed, to avoid copying of unevaluated expressions which would destroy sharing and call-time choice. Notice also that θ may substitute any c-term for the extra variables in the used program rule. The rule **(Contx)** allows to select any subexpression as a redex for the derivation. The rest of the rules are syntactic manipulations of *let*-expressions. In particular **(LetIn)** transforms standard expressions by in-

(Contx)	$\mathcal{C}[e] \to_l \mathcal{C}[e']$, if $e \to_l e'$, $\mathcal{C} \in Cntxt$
(LetIn)	$h(\ldots, e, \ldots) \to_l let\ X = e\ in\ h(\ldots, X, \ldots)$ if $h \in CS \cup FS$, e takes one of the forms $e \equiv f(\overline{e'})$ with $f \in FS^n$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable
(Flat)	$let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \ \to_l\ let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ assuming that Y does not appear free in e_3
(Bind)	$let\ X = t\ in\ e \ \to_l\ e[X/t]$, if $t \in CTerm$
(Elim)	$let\ X = e_1\ in\ e_2 \to_l e_2$, if X does not appear free in e_2
(Fapp)	$f(t_1\theta, \ldots, t_n\theta) \ \to_l\ e\theta$, if $f(t_1, \ldots, t_n) \to e \in \mathcal{P}$, $\theta \in CSubst$

Fig. 1. Rules of *let*-rewriting

$heads(repeat(coin)) \to_l$	(**LetIn**)
$let\ X = repeat(coin)\ in\ heads(X) \to_l$	(**LetIn**)
$let\ X = (let\ Y = coin\ in\ repeat(Y))\ in\ heads(X) \to_l$	(**Flat**)
$let\ Y = coin\ in\ let\ X = repeat(Y)\ in\ heads(X) \to_l$	(**Fapp**)
$let\ Y = 0\ in\ let\ X = repeat(Y)\ in\ heads(X) \to_l$	(**Bind**)
$let\ X = repeat(0)\ in\ heads(X) \to_l$	(**Fapp**)
$let\ X = 0 : repeat(0)\ in\ heads(X) \to_l$	(**LetIn**)
$let\ X = (let\ Z = repeat(0)\ in\ 0 : Z)\ in\ heads(X) \to_l$	(**Flat**)
$let\ Z = repeat(0)\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	(**Fapp**)
$let\ Z = 0 : repeat(0)\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l^*$	(**LetIn**), (**Flat**)
$let\ U = repeat(0)\ in\ let\ Z{=}0{:}U\ in\ let\ X{=}0{:}Z\ in\ heads(X) \to_l^*$	(**Bind**)$^{\mathbf{2}}$
$let\ U = repeat(0)\ in\ heads(0 : 0 : U) \to_l$	(**Fapp**)
$let\ U = repeat(0)\ in\ (0, 0) \to_l$	(**Elim**)
$(0, 0)$	

Fig. 2. A let-rewriting derivation

troducing a *let*-binding to express sharing. On the other hand, **(Bind)** removes a *let*-construction for a variable when its binding expression has been evaluated. **(Elim)** allows to remove a binding when the variable does not appear in the body of the construction, which means that the corresponding value is not needed for evaluation. This rule is needed because the expected normal forms are c-terms not containing *let*s. **(Flat)** is needed for flattening nested *let*s, otherwise some reductions could become wrongly blocked or forced to diverge (see [15]). Figure 2 contains a *let*-rewriting derivation for the expression *heads(repeat(coin))* using the program example of Sect. 1. At each step we indicate the used rule (in many occasions in combination with **(Contx)**, but this is not made explicit).

3 *Let*-Narrowing

It is well known that in functional logic computations there are situations where rewriting is not enough, and must be lifted to some kind of *narrowing*, because the expression being reduced contains variables for which different bindings might produce different evaluation results. This might happen either because variables are already present in the initial expression to reduce, or due to the presence of extra variables in the program rules. In the latter case *let*-rewriting certainly works, but not in an effective way, since the parameter passing substitution in the rule **(Fapp)** of Figure 1 'magically' guesses the appropriate values for those extra variables (see Example 2). Although some works [3,5,4] have proved that this process of guessing can be replaced by a systematic non-deterministic generation of all (ground) possible values, the procedure does not cover all aspects of narrowing, since narrowing is able to produce non-ground answers, while generator functions are not.

The standard definition of *narrowing* as a lifting of rewriting in ordinary TRS says (adapted to the notation of contexts): $\mathcal{C}[f(\overline{t})] \rightsquigarrow_\theta \mathcal{C}\theta[r\theta]$, if θ is a mgu of $f(\overline{t})$ and $f(\overline{s})$, where $f(\overline{s}) \to r$ is a fresh variant of a rule of the TRS. We note that frequently the narrowing step is not decorated with the whole unifier θ, but with its projection over the variables in the narrowed expression. The condition that the binding substitution θ is a mgu can be relaxed to accomplish with certain narrowing strategies like needed narrowing [2], which use unifiers but not necessarily most general ones.

This definition of narrowing cannot be directly translated as it is to the case of *let*-rewriting, for two important reasons. The first is not new: because of call-time choice, binding substitutions must be c-substitutions, as already happened in *let*-rewriting. The second is that produced variables (those introduced by **(LetIn)** and bound in a *let* construction) should not be narrowed, because their role is to express intermediate values that are evaluated at most once and shared, according to call-time choice. Therefore the value of produced variables should be better obtained by evaluation of their binding expressions, and not by bindings coming from narrowing steps. Furthermore, to narrow on produced variables destroys the structure of *let*-expressions.

The following example illustrates some of the points above.

Example 1. Consider the following program over natural numbers (represented with constructors 0 and s):

$$\begin{array}{ll}
0 + Y \rightarrow Y & even(X) \rightarrow if\ (Y{+}Y == X)\ then\ true \\
s(X) + Y \rightarrow s(X + Y) & if\ true\ then\ Y \rightarrow Y \\
0 == 0 \rightarrow true & s(X) == s(Y) \rightarrow X == Y \\
0 == s(Y) \rightarrow false & s(X) == 0 \rightarrow false \\
coin \rightarrow 0 & coin \rightarrow s(0)
\end{array}$$

Notice that the rule for *even* has an extra variable Y. With this program, the evaluation of *even(coin)* by *let*-rewriting could start as follows:

$$\begin{array}{l}
even(coin) \rightarrow_l let\ X = coin\ in\ even(X) \\
\rightarrow_l let\ X = coin\ in\ if\ (Y + Y == X)\ then\ true \\
\rightarrow_l^* let\ X = coin\ in\ let\ U = Y + Y\ in\ let\ V = (U == X)\ in\ if\ V\ then\ true \\
\rightarrow_l^* let\ U = Y + Y\ in\ let\ V = (U == 0)\ in\ if\ V\ then\ true
\end{array}$$

Now, all function applications involve variables and therefore narrowing is required to continue the evaluation. But notice that if we perform classical narrowing in (for instance) *if V then true*, then the binding $\{V/true\}$ is created. What to do with this binding in the surrounding context? If the binding is textually propagated to the whole expression, we obtain

$$let\ U{=}Y{+}Y\ in\ let\ true{=}(U{=}{=}0)\ in\ true$$

which is not a legal expression of *LExp* (due to the binding *let true =(U==0)*). If, as it could seem more correct by our variable convention, we rename the V occurring as produced in the the context, we would obtain

$$let\ U{=}Y{+}Y\ in\ let\ W{=}(U{=}{=}0)\ in\ true$$

losing the connection between V and *true*; as a result, the expression above reduces now to *true* only 'by chance', and the same result would have been obtained also if *coin* had been reduced to $s(0)$ instead of 0, which is obviously wrong.

A similar situation would arise if narrowing was done in $U == 0$, giving the binding $\{U/0\}$. The problem in both cases stems from the fact that V, U are bound variables. What is harmless is to perform narrowing in $Y + Y$ (Y is a free variable). This gives the binding $\{Y/0\}$ and the result 0 for the subexpression $Y + Y$. Put in its surrounding context, the derivation above would continue as follows:

$$\begin{array}{l}
let\ U = 0\ in\ let\ V = (U == 0)\ in\ if\ V\ then\ true \\
\rightarrow_l let\ V = (0 == 0)\ in\ if\ V\ then\ true \\
\rightarrow_l let\ V = true\ in\ if\ V\ then\ true \\
\rightarrow_l if\ true\ then\ true \rightarrow_l true
\end{array}$$

The previous example shows that *let*-narrowing *must protect produced variables* against bindings. To express this we could add to the narrowing relation a parameter containing the set of protected variables. Instead of that, we have found more convenient to consider a distinguished set $PVar \subset \mathcal{V}$ of *produced variables* notated as $X_p, Y_p, \ldots$, to be used according to the following criteria: variables bound in a *let* expression must belong to $PVar$ (therefore *let* expressions have the form *let* X_p*=e in e'*); program rules (and fresh variants of them) do not use variables of $PVar$; the parameter passing c-substitution θ in the rule **(Fapp)** of *let*-rewriting replaces extra variables in the rule by c-terms not having variables of $PVar$; and rewriting (or narrowing) sequences start with initial expressions e not having free occurrences of produced variables (i.e., $FV(e) \cap PVar = \emptyset$). Furthermore we will need the following notion:

Definition 1 (Admissible substitutions). *A substitution θ is called* admissible *iff $\theta \in CSubst$ and $(dom(\theta) \cup ran(\theta)) \cap PVar = \emptyset$.*

Admissible substitutions do not interfere with produced variables, and play a role in the results relating *let*-narrowing and *let*-rewriting that can be found below. A nice property of admissible substitutions is that $(\mathcal{C}[e])\theta \equiv \mathcal{C}\theta[e\theta]$ if θ is admissible, but the identity can fail otherwise: consider for instance $\mathcal{C} \equiv$ *let* $X_p = 0$ *in* $[\ ]$, $e \equiv X_p$, $\theta \equiv \{X_p/0\}$; then $(\mathcal{C}[e])\theta \equiv$ *let* $Y_p = 0$ *in* Y_p, but $\mathcal{C}\theta[e\theta] \equiv$ *let* $Y_p = 0$ *in* 0.

The one-step *let*-narrowing relation $e \leadsto^l_\theta e'$ (assuming a given program $\mathcal{P}$) is defined in Figure 3. The rules *Elim, Bind, Flat, LetIn* of *let*-rewriting are kept untouched except for the decoration with the empty substitution ϵ.

(Contx) $\mathcal{C}[e] \leadsto^l_\theta \mathcal{C}\theta[e']$ if $e \leadsto^l_\theta e'$, $\mathcal{C} \in Cntxt$
(Narr) $f(\overline{t}) \leadsto^l_{\theta|_{FV(f(\overline{t}))}} r\theta$, for any fresh variant $(f(\overline{p}) \rightarrow r) \in \mathcal{P}$ and $\theta \in CSubst$ such that:
i) $f(\overline{t})\theta \equiv f(\overline{p})\theta$.
ii) $dom(\theta) \cap PVar = \emptyset$.
iii) $ran(\theta|_{\setminus FV(f(\overline{p}))}) \cap PVar = \emptyset$.
(X) $e \leadsto^l_\epsilon e'$ if $e \rightarrow_l e'$ using $\boldsymbol{X} \in \{Elim, Bind, Flat, LetIn\}$.

Fig. 3. Rules of *let*-narrowing

The rule **(Narr)** requires some explanations:

- We impose $\theta \in CSubst$ to ensure that call-time choice is respected, as in the rule **(Fapp)** of *let*-rewriting.
- The condition *(i)* simply expresses that θ is a unifier of $f(\overline{t})$ and $f(\overline{p})$. To avoid unnecessary loss of generality or applicability of our approach, we do not impose θ to be a mgu. We stress the fact that Th. 1 guarantees soundness of this 'liberal' *let*-narrowing (which implies soundness of *let*-narrowing restricted to use only mgu's), and completeness is stated and proved in Th. 2 for *let*-narrowing using mgu's (which implies completeness of unrestricted *let*-narrowing).

- The condition *(ii)* expresses the protection of produced variables against narrowing justified in Example 1.
- Condition *(iii)* is a subtle one stating that the bindings in θ for extra variables and for variables in the expression being narrowed do not introduce produced variables. Otherwise, undesired situations arise, when **(Narr)** is combined with **(Contx)**. Consider for instance, the program rules $f \rightarrow Y$ and $loop \rightarrow loop$ and the expression $let\ X_p = loop\ in\ f$. This expression could be reduced in the following way:

  $$let\ X_p = loop\ in\ f \rightsquigarrow^l_\epsilon let\ X_p = loop\ in\ Z$$

  by applying **(Narr)** to f with $\theta = \epsilon$ taking the fresh variant rule $f \rightarrow Z$, and using **(Contx)** for the whole expression. If we drop condition (*iii*) we could perform a similar derivation using the same fresh variant of the rule for f, but now using the substitution $\theta = \{Z/X_p\}$:

  $$let\ X_p = loop\ in\ f \rightsquigarrow^l_\epsilon let\ X_p = loop\ in\ X_p$$

  which is certainly not intended because the free variable Z in the previous derivation appears now as a produced variable, i.e., we get an undesired capture of variables.

  On the other hand, we also remark that if the substitution θ in **(Narr)** is chosen to be a standard mgu[2] of $f(\overline{t})$ and $f(\overline{p})$ (which is always possible) then the condition *(iii)* is always fulfilled.
- Notice finally that in a **(Narr)**-step not the whole θ is recorded, but only its projection over the relevant variables, i.e. the variables in the narrowed expression. This, together with *(i)* and *(ii)*, guarantees that the annotated substitution is an admissible one, a property not fulfilled in general by the whole θ, since the parameter passing (one of the roles of θ) might bind variables coming from the program rule to terms containing produced variables (as for example with the program $\{f(X) \rightarrow 0, loop \rightarrow loop\}$ and the step $let\ X_p = loop\ in\ f(X_p) \rightsquigarrow^l_\epsilon let\ X_p = loop\ in\ 0$, which uses $\theta = \{X/X_p\}$).

Regarding the rule **(Contx)**, notice that θ is either ϵ or is obtained by **(Narr)** applied to the inner e. By the conditions imposed over unifiers in **(Narr)**, θ does not bind any produced variable, including those in *let* expressions surrounding e, which guarantees that any piece of the form $let\ X_p = r\ in\ \ldots$ occurring in $\mathcal{C}$ becomes $let\ X_p = r\theta\ in\ \ldots$ in $\mathcal{C}\theta$ after the narrowing step. Furthermore, conditions of **(Narr)** ensure also that no produced variable capture is generated. The one-step relation $\rightsquigarrow^l_\theta$ is extended in the natural way to the multiple-steps narrowing relation $\rightsquigarrow^{l^*}$, which is defined as the least reflexive relation verifying:

$$e \rightsquigarrow^l_\epsilon e \qquad e \rightsquigarrow^l_{\theta_1} e_1 \rightsquigarrow^l_{\theta_2} \ldots e_n \rightsquigarrow^l_{\theta_n} e' \;\Rightarrow\; e \rightsquigarrow^{l^*}_{\theta_1 \ldots \theta_n} e'$$

We write $e \rightsquigarrow^{l^n}_\theta e'$ for a n-steps narrowing sequence. Observe that with the given definitions of one-step and multiple-step narrowing, $e \rightsquigarrow^l_\theta e'$ implies $dom(\theta) \subseteq$

[2] By standard mgu of t, s we mean an idempotent mgu θ with $dom(\theta) \cup ran(\theta) \subseteq var(t) \cup var(s)$.

$$\begin{array}{ll}
\mathit{even}(\mathit{coin}) \rightsquigarrow^{l}_{\epsilon} & (\mathbf{LetIn}) \\
\mathit{let}\ X_p = \mathit{coin}\ \mathit{in}\ \underline{\mathit{even}(X_p)} \rightsquigarrow^{l}_{\epsilon} & (\mathbf{Narr}) \\
\mathit{let}\ X_p = \mathit{coin}\ \mathit{in}\ \mathit{if}\ Y+Y == X_p\ \mathit{then}\ \mathit{true} \rightsquigarrow^{l^3}_{\epsilon} & (\mathbf{LetIn})^{\mathbf{2}}, (\mathbf{Flat}) \\
\mathit{let}\ X_p = \underline{\mathit{coin}}\ \mathit{in}\ \mathit{let}\ U_p = Y+Y\ \mathit{in} & \\
\quad \mathit{let}\ V_p = (U_p == X_p)\ \mathit{in}\ \mathit{if}\ V_p\ \mathit{then}\ \mathit{true} \rightsquigarrow^{l}_{\epsilon} & (\mathbf{Narr}) \\
\mathit{let}\ X_p = 0\ \mathit{in}\ \mathit{let}\ U_p = Y+Y\ \mathit{in} & \\
\quad \mathit{let}\ V_p = (U_p == X_p)\ \mathit{in}\ \mathit{if}\ V_p\ \mathit{then}\ \mathit{true} \rightsquigarrow^{l}_{\epsilon} & (\mathbf{Bind}) \\
\mathit{let}\ U_p = \underline{Y+Y}\ \mathit{in}\ \mathit{let}\ V_p = (U_p == 0)\ \mathit{in}\ \mathit{if}\ V_p\ \mathit{then}\ \mathit{true} \rightsquigarrow^{l}_{\{Y/0\}} & (\mathbf{Narr}) \\
\mathit{let}\ U_p = 0\ \mathit{in}\ \mathit{let}\ V_p = (U_p == 0)\ \mathit{in}\ \mathit{if}\ V_p\ \mathit{then}\ \mathit{true} \rightsquigarrow^{l}_{\epsilon} & (\mathbf{Bind}) \\
\mathit{let}\ V_p = \underline{(0 == 0)}\ \mathit{in}\ \mathit{if}\ V_p\ \mathit{then}\ \mathit{true} \rightsquigarrow^{l}_{\epsilon} & (\mathbf{Narr}) \\
\mathit{let}\ V_p = \mathit{true}\ \mathit{in}\ \mathit{if}\ V_p\ \mathit{then}\ \mathit{true} \rightsquigarrow^{l}_{\epsilon} & (\mathbf{Bind}) \\
\mathit{if}\ \mathit{true}\ \mathit{then}\ \mathit{true} \rightsquigarrow^{l}_{\epsilon} & (\mathbf{Narr}) \\
\mathit{true} &
\end{array}$$

Fig. 4. A *let*-narrowing derivation

$FV(e)$, but this is not necessarily true for $e \rightsquigarrow^{l^*}_{\theta} e'$, since narrowing with a program rule with extra variables may introduce new free variables, whose bindings in later narrowing steps would be recorded and collected in the overall $e \rightsquigarrow^{l^*}_{\theta} e'$. However, to project the accumulated θ over $FV(e)$, as is done in one step, does not cause any essential harm and would require only minor changes in our results and proofs.

Example 2. Example 1 essentially contains already a narrowing derivation. For the sake of clarity, we repeat it in Figure 4 making explicit the rule of *let*-narrowing used at each step (maybe in combination with **(Contx)**, which is not written). Besides, if the step uses **(Narr)**, the narrowed expression is underlined. Notice that all **(Narr)** steps in the derivation except one have ϵ as narrowing substitution (because of the projection over the variables of the narrowed expression), so they are really rewriting steps. An additional remark that could help to further explain the relation between *let*-narrowing and *let*-rewriting is the following: since we have $\mathit{even}(\mathit{coin}) \rightsquigarrow^{l}_{\theta}\ \mathit{true}$ for some θ, but $\mathit{even}(\mathit{coin})$ is ground, Th. 1 in next section ensures that there must be also a successful *let*-rewriting derivation $\mathit{even}(\mathit{coin}) \rightarrow^{*}_{l}\ \mathit{true}$. This derivation could have the form:

$$\begin{array}{ll}
\mathit{even}(\mathit{coin}) \rightarrow_l & (\mathbf{LetIn}) \\
\mathit{let}\ X_p = \mathit{coin}\ \mathit{in}\ \mathit{even}(X_p) \rightarrow_l & (\mathbf{Fapp}) \\
\mathit{let}\ X_p = \mathit{coin}\ \mathit{in}\ \mathit{if}\ (0+0 == X_p)\ \mathit{then}\ \mathit{true} \rightarrow_l & \\
\ldots\ldots\ldots \rightarrow_l\ \mathit{true} &
\end{array}$$

The indicated **(Fapp)**-step in this *let*-rewriting derivation has used the substitution $\{Y/0\}$, thus anticipating and 'magically guessing' the right value of the extra variable Y of the rule of *even*. In contrast, in the *let*-narrowing derivation the binding for Y is not done while reducing $\mathit{even}(X)$ but in a later **(Narr)**-step over $Y+Y$. This corresponds closely to the behavior of systems narrowing-based systems like Curry or Toy.

3.1 Soundness and Completeness of *let*-Narrowing

In this section we show the adequacy of *let*-narrowing wrt *let*-rewriting. We assume a fixed program $\mathcal{P}$. The following useful property of let-rewriting will be employed in some proofs:

Lemma 1 (Closedness of $\rightarrow_l$ under c-substitutions). *For any $e, e' \in LExp$ and $\theta \in CSubst$ we have $e \rightarrow_l^* e'$ implies $e\theta \rightarrow_l^* e'\theta$.*

Proof. A simple induction on the length of the derivation $e \rightarrow_l^* e'$ (see [20] for details).

Now we are ready to prove *soundness*, as stated in the following result:

Theorem 1 (Soundness of *let*-narrowing). *For any $e, e' \in LExp$, $e \leadsto_\theta^{l^*} e'$ implies $e\theta \rightarrow_l^* e'$.*

Proof. We first prove soundness of one step of narrowing, that is: $e \leadsto_\theta^l e'$ implies $e\theta \rightarrow_l e'$. We proceed by a case distinction over the applied rule of *let*-narrowing. The cases of **(Elim)**, **(Bind)**, **(Flat)** and **(LetIn)** are trivial, since narrowing and rewriting coincide for these rules. For **(Narr)**, assume $f(\overline{t}) \leadsto^l_{\theta|_{FV(f(\overline{t}))}} r\theta$ with $(f(\overline{t}))\theta \equiv (f(\overline{p}))\theta$; then we have $(f(\overline{t}))\theta|_{FV(f(\overline{t}))} \equiv (f(\overline{t}))\theta \equiv (f(\overline{p}))\theta$ and we can perform a *let*-rewriting step $(f(\overline{t}))\theta|_{FV(f(\overline{t}))} \rightarrow_l r\theta$, by **(Fapp)**, using θ as matching substitution.

For the rule **(Contx)**, assume $\mathcal{C}[e] \leadsto_\theta^l \mathcal{C}\theta[e']$ with $e \leadsto_\theta^l e'$, where the step $e \leadsto_\theta^l e'$ does not involve again the rule **(Contx)** (notice that several nested applications of **(Contx)** can be reduced to a unique application of such a rule just choosing the appropriate context). Since we have proved the rest of the cases we get $e\theta \rightarrow_l e'$. Since θ is admissible as it is ϵ or comes from a **(Narr)**-step, we have $(\mathcal{C}[e])\theta \equiv \mathcal{C}\theta[e\theta]$. Therefore, using the rule **(Contx)** of *let*-rewriting we obtain $(\mathcal{C}[e])\theta \equiv \mathcal{C}\theta[e\theta] \rightarrow_l \mathcal{C}\theta[e']$.

For the case of several steps $e \leadsto_\theta^{l^n} e'$, we proceed by induction over the number of steps n. If $n = 0$ we have that $e \leadsto_\epsilon^{l^0} e$, but also $e \rightarrow_l^0 e$. If $n > 0$ we will have $e \leadsto_\sigma^l e'' \leadsto_\gamma^{l^{n-1}} e'$ with $\theta = \sigma\gamma$. Because of the just proved soundness of one step of narrowing, we have $e\sigma \rightarrow_l e''$ and therefore $e\sigma\gamma \rightarrow_l e''\gamma$, because $\rightarrow_l$ is closed under c-substitutions (Lemma 1). But then, since $e''\gamma \rightarrow_l^* e'$ by the induction hypothesis, we conclude $e\theta \equiv e\sigma\gamma \rightarrow_l^* e'$, as desired. □

Completeness is, as usual, more complicated to prove. The key result is the following generalization of Hullot's *lifting lemma* for classical rewriting and narrowing [13], to the framework of *let*-rewriting. This lemma states that any rewrite sequence for a particular instance of an expression can be generalized by a narrowing derivation.

Lemma 2 (Lifting lemma for *let*-rewriting). *Let $e, e' \in LExp$ such that $e\theta \rightarrow_l^* e'$ for an admissible θ, and let $\mathcal{W}$ be a finite set of variables with $dom(\theta) \cup FV(e) \subseteq \mathcal{W}$. Then there exist a* let*-narrowing derivation $e \leadsto_\sigma^{l^*} e''$ and an admissible θ' such that $e''\theta' = e'$ and $\sigma\theta' = \theta[\mathcal{W}]$. Besides, the* let*-narrowing derivation can be chosen to use mgu's at each* **(Narr)** *step. Graphically:*

$$\begin{tikzcd}
e \arrow[r, rightsquigarrow, dotted, "l^*", "\sigma"'] \arrow[d, mapsto, "\theta"'] & e'' \arrow[d, mapsto, dashed, "\theta'"] \\
e\theta \arrow[r, "*", "l"'] & e'
\end{tikzcd}$$

Proof. We first prove the result for one-step derivations $e\theta \to_l e'$. We distinguish cases depending on the rule of *let*-rewriting applied for the step.

$\boxed{\mathbf{X}}$ All the cases included here (**Elim**,**Bind**,**Flat**,**LetIn**) can be proved easily taking $\sigma = \epsilon$, $\theta' = \theta$, and performing the *let*-narrowing step with the same **X** rule.

$\boxed{\mathbf{Fapp}}$ Then $e \equiv f(t_1, \ldots, t_n)$ and the step given was $e\theta \equiv f(t_1\theta, \ldots, t_n\theta) \to_l s\gamma$ $(f(p_1, \ldots, p_n) \to s) \in \mathcal{P}$ and $\gamma \in CSubst$ such that $f(\overline{p})\gamma \equiv f(\overline{t})\theta$. We will see how to build the following diagram:

$$\begin{tikzcd}
e \equiv f(\overline{t}) \arrow[r, rightsquigarrow, dotted, "l^*", "\sigma"'] \arrow[d, mapsto, "\theta"'] & e'' \arrow[d, mapsto, dashed, "\theta'"] \\
f(\overline{t})\theta \arrow[r, "*", "l"'] & s\gamma \equiv e'
\end{tikzcd}$$

Without loss of generality we can suppose that the variant of the program rule used in the step is fresh and $dom(\gamma) \subseteq FV(f(\overline{p}) \to s)$, so $\gamma \uplus \theta$ is correctly defined and it is a unifier of $f(\overline{p})$ and $f(\overline{t})$. But then there must exist $\sigma_0 = mgu(\overline{p}, \overline{t})$ with $dom(\sigma_0) \subseteq dom(\gamma \uplus \theta)$, so $dom(\sigma_0) \cap PVar = \emptyset$ as no $PVar$ is present in the domains of γ (as no $PVar$ is present in a variant of a program rule) and θ (as it is admissible). Besides, $ran(\sigma_0|_{\setminus FV(f(\overline{p}))}) \cap PVar = \emptyset$, as the only produced variables introduced by a mgu are those needed to unify, in this case those in $ran(\sigma_0|_{FV(f(\overline{p}))})$; and $\sigma_0 \in CSubst$ as it is the mgu of a pair of tuples of c-terms. So we can choose $\sigma = \sigma_0|_{FV(f(\overline{t}))}$ to perform $e \equiv f(\overline{t}) \rightsquigarrow^{l^*}_{\sigma} s\sigma_0 \equiv e''$. Now we can define $\theta' = \theta'_0 \uplus \theta'_1$, where:

- As $\gamma \uplus \theta$ is a unifier of $f(\overline{p})$ and $f(\overline{t})$ then $\sigma_0 \preceq (\gamma \uplus \theta)$ and so $\exists \theta'_1 \in CSubst$ such that $\sigma_0\theta'_1 = \gamma \uplus \theta$ and $dom(\theta'_1) \subseteq ran(\sigma_0) \cup (dom(\gamma \uplus \theta) \setminus dom(\sigma_0))$.
- $\theta'_0 = \theta|_{\setminus(dom(\theta'_1)\ \cup\ FV(f(\overline{t})))}$.

Obviously $dom(\theta'_0) \cap dom(\theta'_1) = \emptyset$, hence θ' is correctly defined. Besides $dom(\theta'_0) \cap (FV(f(\overline{t})) \cup FV(f(\overline{p}))) = \emptyset$, as if $X \in FV(f(\overline{t}))$ then $X \notin dom(\theta'_0)$ by definition; and if $X \in FV(f(\overline{p}))$ then it belongs to the fresh variant and so $X \notin dom(\theta) \supseteq dom(\theta'_0)$. Now it can be shown that θ' is admissible.

- $e''\theta' \equiv s\sigma_0\theta' \equiv s\sigma_0\theta'_1$ because given $X \in FV(s\sigma_0)$ if $X \in FV(s)$ then it belongs to the fresh variant and so $X \notin dom(\theta) \supseteq dom(\theta'_0)$; and if $X \in ran(\sigma_0) \subseteq$

$FV(f(\overline{t})) \cup FV(f(\overline{p}))$ (as $\sigma_0 = mgu(\overline{p}, \overline{t})$) then $X \notin dom(\theta'_0)$ because $dom(\theta'_0) \cap (FV(f(\overline{t})) \cup FV(f(\overline{p}))) = \emptyset$. But $s\sigma_0\theta'_1 \equiv s(\gamma \uplus \theta) \equiv s\gamma \equiv e'$, because $\sigma_0\theta'_1 = \gamma \uplus \theta$ and s is part of the fresh variant.
- $\sigma\theta' = \theta[\mathcal{W}]$: Given $X \in \mathcal{W}$, if $X \in FV(f(\overline{t}))$ then $X \notin dom(\gamma)$ and so $\theta(X) \equiv (\gamma \uplus \theta)(X) \equiv \sigma_0\theta'_1(X) = \sigma\theta'_1(X)$, as $\sigma\theta'_1 = \theta \uplus \gamma$ and $\sigma = \sigma_0|_{FV(\overline{t})}$. But $\sigma\theta'_1(X) \equiv \sigma\theta'(X)$ because given $Z \in FV(\sigma(X))$, if $Z \equiv X$ then as $X \in FV(f(\overline{t}))$ then $Z \equiv X \notin dom(\theta'_0)$ by definition of θ'_0; if $Z \in ran(\sigma) \subseteq ran(\sigma_0)$ then $Z \notin dom(\theta'_0)$, as we saw before.

On the other hand, $(\mathcal{W} \setminus FV(f(\overline{t}))) \cap (FV(f(\overline{t})) \cup FV(f(\overline{p}))) = \emptyset \cup \emptyset = \emptyset$, because $FV(f(\overline{p}))$ are part of the fresh variant. So, if $X \in \mathcal{W} \setminus FV(f(\overline{t}))$, then $X \notin dom(\sigma) \subseteq dom(\sigma_0) \subseteq FV(f(\overline{t})) \cup FV(f(\overline{p}))$. Now if $X \in dom(\theta'_0)$ then $\theta(X) \equiv \theta'_0(X)$ (by definition of θ'_0), $\theta'_0(X) \equiv \theta'(X)$ (as $X \in dom(\theta'_0)$), $\theta'(X) \equiv \sigma\theta'(X)$ (as $X \notin dom(\sigma)$). If $X \in dom(\theta'_1)$, $\theta(X) \equiv (\gamma \uplus \theta)(X)$ (as $X \in \mathcal{W} \setminus FV(f(\overline{t}))$ implies it does not appear in the fresh instance), $(\gamma \uplus \theta)(X) \equiv \sigma_0\theta'_1(X)$ (as $\sigma_0\theta'_1 = \gamma \uplus \theta$), $\sigma_0\theta'_1(X) \equiv \theta'_1(X)$ (as $X \notin dom(\sigma_0)$), $\theta'_1(X) \equiv \theta'(X)$ (as $X \in dom(\theta'_1)$) and $\theta'(X) \equiv \sigma\theta'(X)$ (as $X \notin dom(\sigma)$). And if $X \notin (dom(\theta'_0) \cup dom(\theta'_1))$ then $X \notin dom(\theta')$, and as $X \notin dom(\sigma)$, $X \notin dom(\sigma_0)$ and $\theta(X) \equiv (\gamma \uplus \theta)(X)$, then $\theta(X) \equiv (\theta \uplus \gamma)(X) \equiv \sigma_0\theta'_1(X) \equiv \theta'_1(X) \equiv X \equiv \theta'(X) \equiv \sigma\theta'(X)$.

$\boxed{\textbf{Contx}}$ Then:

- $e \equiv \mathcal{C}[s]$ for some $s \in LExp$.
- $e\theta \equiv (\mathcal{C}[s])\theta \equiv \mathcal{C}\theta[s\theta]$, as θ is admissible.

So the step was $e\theta \equiv \mathcal{C}\theta[s\theta] \to_l \mathcal{C}\theta[s'] \equiv e'$ with $s\theta \to_l s'$ a step where **(Contx)** was not applied (as usual we can assume this taking a suitable context). Then, by the proof of the other cases and taking $\mathcal{W}' = \mathcal{W} \cup FV(s)$ we get $s \leadsto^{l^*}_{\sigma_2} s''$ such that there exist an admissible θ'_2 with $s''\theta_2 \equiv s'$ and $\sigma_2\theta'_2 = \theta[\mathcal{W}']$. But then $e \equiv \mathcal{C}[s] \leadsto^{l}_{\sigma_2} \mathcal{C}\sigma_2[s''] \equiv e''$ and we can take $\sigma = \sigma_2$ and $\theta' = \theta'_2$ getting:
- θ' is admissible as θ'_2 is.
- As $\sigma_2\theta'_2 = \theta[\mathcal{W}']$ and by construction $\mathcal{W} \subseteq \mathcal{W}'$, then $\sigma_2\theta'_2 = \theta[\mathcal{W}]$ and so $(\sigma\theta')|_{\mathcal{W}} = (\sigma_2\theta'_2)|_{\mathcal{W}} = \theta|_{\mathcal{W}}$.
- $e''\theta' \equiv (\mathcal{C}\sigma_2[s''])\theta'_2 \equiv \mathcal{C}\sigma_2\theta'_2[s''\theta'_2]$, as θ'_2 is admissible, and $\mathcal{C}\sigma_2\theta'_2[s''\theta'_2] \equiv \mathcal{C}\theta[s'] \equiv e'$, as $s''\theta_2 \equiv s'$ and $\sigma_2\theta'_2 = \theta[\mathcal{W}]$, $FV(\mathcal{C}[]) \subseteq \mathcal{W}$.

We have now finished the proof for one-step *let*-rewriting derivations. To generalize the proof to any number of steps we proceed by induction on the length n of $e\theta \to^n_l e'$. In the base case $n = 0$ and so we have $e\theta \to^0_l e\theta \equiv e'$, so we can do $e \leadsto^l_\epsilon e \equiv e''$ getting $\sigma = \epsilon$, and taking $\theta' = \theta$. So $e''\theta' \equiv e\theta \equiv e'$ and $\sigma\theta' = \epsilon\theta = \theta$.

For more than one step we start with $e\theta \to_l e_1 \to^*_l e'$. Then using the proof for one step we get $e \leadsto^l_{\sigma_1} e''_1$ and θ'_1 admissible with $e''_1\theta'_1 \equiv e_1$ and $\sigma_1\theta'_1 = \theta[\mathcal{W}]$. Now if we define $\mathcal{U}_1 = (\mathcal{W} \setminus dom(\sigma_1)) \cup ran(\sigma_1) \cup vE$, with vE the set containing the extra variables in the program rule applied in $e\theta \to_l e_1$, or $\emptyset$ if (Fapp) was not applied, it can be proved that $FV(e''_1) \subseteq \mathcal{U}_1$. Besides, if we define $\theta_1 = \theta'_1|_{\mathcal{U}_1}$ then it can be shown that $\sigma_1\theta_1 = \theta[\mathcal{W}]$ and $e''_1\theta_1 \equiv e_1$, so $e''_1\theta_1 \equiv e_1 \to^*_l e'$ and we can apply HI to this derivation using $\mathcal{U}_1$ getting $e''_1 \leadsto^{l^*}_{\sigma_2} e''_2$

$$\begin{tikzcd}
e \arrow[r, squiggly, dotted, "l", "\sigma_1"'] \arrow[d, mapsto, "\theta"'] & e_1'' \arrow[r, squiggly, dotted, "l^*", "\sigma_2"'] \arrow[d, dashed, mapsto, "\theta_1"'] & e_2'' \equiv e'' \arrow[d, dashed, mapsto, "\theta_2 = \theta'"] \\
e\theta \arrow[r, "l"'] & e_1 \arrow[r, "*", "l"'] & e'
\end{tikzcd}$$

with admissible θ_2, $e_2''\theta_2 \equiv e'$ and $\sigma_2\theta_2 = \theta_1[\mathcal{U}_1]$. But then $e \leadsto^{l^*}_{\sigma_1\sigma_2} e_2''$ so we can use $\sigma = \sigma_1\sigma_2$, $\theta' = \theta_2$, $e'' \equiv e_2''$ getting:

- • $\theta' = (\theta_2)$ is admissible, and $e''\theta' \equiv e_2''\theta_2 \equiv e'$.
- • $\sigma\theta' = \theta[\mathcal{W}]$, that is $\sigma_1\sigma_2\theta' = \theta[\mathcal{W}]$. As $\sigma_1\theta_1 = \theta[\mathcal{W}]$ all that is left is proving $\sigma_1\sigma_2\theta_2 = \sigma_1\theta_1[\mathcal{W}]$, and this is indeed the case as:
  - if $X \in dom(\sigma_1)$ then $FV(\sigma_1(X)) \subseteq ran(\sigma_1) \subseteq \mathcal{U}_1$, so as $\sigma_2\theta_2 = \theta_1[\mathcal{U}_1]$ then $(\sigma_1(X))\sigma_2\theta_2 \equiv (\sigma_1(X))\theta_1$.
  - if $X \in \mathcal{W} \setminus dom(\sigma_1)$ then $X \in \mathcal{U}_1$, so as $\sigma_2\theta_2 = \theta_1[\mathcal{U}_1]$ then $(\sigma_1(X))\sigma_2\theta_2 \equiv X\sigma_2\theta_2 \equiv X\theta_1 \equiv X\sigma_1\theta_1$. □

Now, combining this result with Th. 1 we obtain a strong adequacy theorem for *let*-narrowing with respect to *let*-rewriting. The left to right implication of this result corresponds to the completeness of *let*-narrowing wrt *let*-rewriting.

Theorem 2 (Adequacy of *let*-narrowing)
Let $e, e_1 \in LExp$ and θ an admissible c-substitution, then:

$$e\theta \to_l^* e_1 \;\Leftrightarrow\; \begin{array}{l} \textit{there exists a let-narrowing derivation } e \leadsto^{l^*}_{\sigma} e_2 \\ \textit{and an admissible } \theta' \textit{ such that } \sigma\theta' = \theta[FV(e)],\ e_2\theta' \equiv e_1 \end{array}$$

Proof

($\Rightarrow$) Assume $e\theta \to_l^* e_1$. As $e\theta|_{FV(e)} \equiv e\theta$ then trivially $e\theta|_{FV(e)} \to_l^* e_1$. We can apply Lemma 2 taking $\mathcal{W} = FV(e)$ to get $e \leadsto^{l^*}_{\sigma} e_2$ such that there exists $\theta' \in CSubst$ with $\sigma\theta' = \theta|_{FV(e)}[\mathcal{W}]$ and $e_2\theta' \equiv e_1$. But as $\mathcal{W} = FV(e)$ then $\sigma\theta' = \theta|_{FV(e)}[\mathcal{W}]$ implies $\sigma\theta' = \theta[FV(e)]$.
We remark that the lifting lemma ensures that the narrowing derivation can be chosen to use mgu's at each **(Narr)** steps.

($\Leftarrow$) Assume $e \leadsto^{l^*}_{\sigma} e_2$ and θ' under the conditions above. Then by Th. 1 we have $e\sigma \to_l^* e_2$. As $\to_l$ is closed under c-substitutions (Lemma 1) then $e\sigma\theta' \to_l^* e_2\theta'$. But as $\sigma\theta' = \theta[FV(e)]$, then $e\theta \equiv e\sigma\theta' \to_l^* e_2\theta' \equiv e_1$. □

3.2 *Let*-Narrowing Versus Narrowing for Deterministic Systems

The relationship between *let*-rewriting ($\to_l$) and ordinary rewriting ($\to$) is examined in [15], where $\to_l$ is proved to be sound wrt $\to$, and complete for the class of *deterministic* programs, a notion close but not equivalent to confluence. Determinism is originally defined in [15] as a semantic notion in the framework of *CRWL*, but an alternative equivalent (see [15], Lemma 13) definition in terms of reduction is the following:

Definition 2. *A program $\mathcal{P}$ is deterministic iff for any expressions $e \in Exp$, $e', e'' \in LExp$ with $e \to_l^* e'$ and $e \to_l^* e''$, there exists $e''' \in LExp$ such that $e \to_l^* e'''$ and $|e'''| \sqsupseteq |e'|, |e'''| \sqsupseteq |e''|$.*

The class of deterministic programs is conjectured in [15] to be wider than that of confluent programs[3] and it certainly contains all inductively sequential programs (without extra variables). The following result holds (see [15]):

Theorem 3. *Let $\mathcal{P}$ be any program, $e \in Exp, t \in CTerm$. Then:*

(a) *$e \to_l^* t$ implies $e \to^* t$.*
(b) *If in addition $\mathcal{P}$ is deterministic, then the reverse implication holds.*

Joining this with the results of the previous section we can easily establish some relationships between *let*-narrowing and ordinary rewriting/narrowing, as follows (we assume here that all involved substitutions are admissible):

Theorem 4. *Let $\mathcal{P}$ be any program, $e \in Exp, \theta \in CSubst, t \in CTerm$. Then:*

(a) *$e \leadsto_\theta^{l^*} t$ implies $e\theta \to^* t$.*
(b) *If in addition $\mathcal{P}$ is deterministic, then:*
 ($\mathbf{b}_1$) *If $e\theta \to^* t$, there exist $t' \in CTerm$, $\sigma, \theta' \in CSubst$ such that $e \leadsto_\sigma^{l^*} t'$, $t'\theta' = t$ and $\sigma\theta' = \theta[var(e)]$ (and therefore $t' \preceq t, \sigma \preceq \theta[var(e)]$).*
 ($\mathbf{b}_2$) *If $e \leadsto_\theta^* t$, the same conclusion of* (b_1) *holds.*

Part *(a)* expresses soundness of $\leadsto^l$ wrt rewriting, and part *(b)* is a completeness result for $\leadsto^l$ wrt rewriting/narrowing, for the class of deterministic programs.

Proof. Part *(a)* follows from soundness of *let*-narrowing wrt *let*-rewriting (Th. 1) and soundness of *let*-rewriting wrt rewriting (part *(a)* of Th. 3).

For *(b_1)*, assume $e\theta \to^* t$. By the completeness of *let*-rewriting wrt rewriting for deterministic programs (part *(b)* of Th. 3), we have $e\theta \to_l^* t$, and then by the completeness of *let*-narrowing wrt *let*-rewriting (Th. 2), there exists a narrowing derivation $e \leadsto_\sigma^{l^*} t'$ with $t'\theta' = t$ and $\sigma\theta' = \theta[FV(e)]$. But notice that for $e \in Exp$, the sets $FV(e)$ and $var(e)$ coincide, and the proof is finished. Finally, *(b_2)* follows simply from soundness of (ordinary) narrowing wrt rewriting and *(b_1)*. □

4 Conclusions

Our aim in this work was to progress in the effort of filling an existing gap in the functional logic programming field, where up to recently there was a lack of a simple and abstract enough one-step reduction mechanism close enough to ordinary rewriting but at the same time respecting non-strict and call-time

[3] It is easy to find examples of deterministic but not confluent programs: this happens, for instance, with the program $\mathcal{P} = \{f \to 0,\ f \to g\}$, where g has no rules. Which is difficult to prove is that confluent programs are always deterministic.

choice semantics for possibly non-confluent and non-terminating constructor-based rewrite systems (possibly with extra variables), and trying to avoid the complexity of graph rewriting [19]. These requirements were not met by two well established branches in the foundations of the field: one is the *CRWL* approach, very adequate from the point of view of semantics but operationally based on somehow complicated narrowing calculi [9,22] too far from the usual notion of term rewriting. The other approach focuses on operational aspects in the form of efficient narrowing strategies like needed narrowing [2] or natural narrowing [7], based on the classical theory of rewriting, sharing with it the major drawback that rewriting is an *unsound* operation for call-time choice semantics of functional logic programs. There have been other attempts of coping operationally with call-time choice [1], but relying in too low-level syntax and operational rules.

In a recent work [15] we established a technical bridge between both approaches (*CRWL*/classical rewriting) by proposing a notion of rewriting with sharing by means of local *let* bindings, in a similar way to what has been done for sharing and lambda-calculus in [18]. Most importantly, we prove there strong equivalence results between *CRWL* and *let*-rewriting.

Here we have continued that work by contributing a notion of *let*-narrowing (narrowing for sharing) which we prove sound and complete with respect to *let*-rewriting. We think that *let*-narrowing is the simplest proposed notion of narrowing that is close to the usual notions of TRS and at the same time is proved adequate for call-time choice semantics. The main technical insight for let-narrowing has been the need of protecting produced (locally bound) variables against narrowing over them. We have also proved soundness of *let*-narrowing wrt ordinary rewriting and completeness for the wide class of deterministic programs, thus giving a technical support to the intuitive fact that combining sharing with narrowing does not create new answers when compared to classical narrowing, and at the same time does not lose answers in case of deterministic systems. As far as we know these results are new in the narrowing literature.

Two natural extensions of our work are: dealing with HO functions (see [16]) and adding strategies to *let*-rewriting and *let*-narrowing, an issue that has been left out of this paper but is needed as foundation of effective implementations. But we think that the clear script we have followed so far (first presenting a notion of rewriting with respect to which we have been able to prove correctness and completeness of a a subsequent notion of narrowing, to which add strategies in future work) is an advantage rather than a lack of our approach.

References

1. Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G.: Operational semantics for declarative multi-paradigm languages. Journal of Symbolic Computation 40(1), 795–829 (2005)
2. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. In: Proc. ACM Symposium on Principles of Programming Languages (POPL 1994), pp. 268–279. ACM Press, New York (1994)
3. Antoy, S., Hanus, M.: Overlapping rules and logic variables in functional logic programs. In: ICLP, pp. 87–101 (2006)

4. Braßel, B., Huch, F.: On a tighter integration of functional and logic programming. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 122–138. Springer, Heidelberg (2007)
5. Dios, J., López-Fraguas, F.: Elimination of extra variables from functional logic programs. In: Lucio, P., Orejas, F. (eds.) VI Jornadas sobre Programación y Lenguajes (PROLE 2006), pp. 121–135. CINME (2006)
6. Echahed, R., Janodet, J.-C.: Admissible graph rewriting and narrowing. In: Proceedings of the Joint International Conference and Symposium on Logic Programming, Manchester, June 1998, pp. 325–340. MIT Press, Cambridge (1998)
7. Escobar, S., Meseguer, J., Thati, P.: Natural narrowing for general term rewriting systems. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 279–293. Springer, Heidelberg (2005)
8. González-Moreno, J.C., Hortalá-González, T., López-Fraguas, F., Rodríguez-Artalejo, M.: A rewriting logic for declarative programming. In: Riis Nielson, H. (ed.) ESOP 1996. LNCS, vol. 1058, pp. 156–172. Springer, Heidelberg (1996)
9. González-Moreno, J.C., Hortalá-González, T., López-Fraguas, F., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. Journal of Logic Programming 40(1), 47–87 (1999)
10. Hanus, M.: Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel (2005)
11. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
12. Hanus, M. (ed.): Curry: An integrated functional logic language (version 0.8.2) (March 2006), `http://www.informatik.uni-kiel.de/report.html`
13. Hullot, J.: Canonical forms and unification. In: Bibel, W. (ed.) CADE 1980. LNCS, vol. 87, pp. 318–334. Springer, Heidelberg (1980)
14. Hussmann, H.: Non-Determinism in Algebraic Specifications and Algebraic Programs. Birkhäuser, Basel (1993)
15. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: A simple rewrite notion for call-time choice semantics. In: Proc. Principles and Practice of Declarative Programming, pp. 197–208. ACM Press, New York (2007)
16. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Rewriting and call-time choice: the HO case. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 147–162. Springer, Heidelberg (2008)
17. López-Fraguas, F., Sánchez-Hernández, J.: $\mathcal{TOY}$: A multiparadigm declarative system. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
18. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. J. Funct. Program. 8(3), 275–317 (1998)
19. Plump, D.: Essentials of term graph rewriting. Electr. Notes Theor. Comput. Sci. 51 (2001)
20. Rodríguez-Hortalá, J.: El indeterminismo en programacin lógico-funcional: un enfoque basado en reescritura. Trabajo de Investigación de Tercer Ciclo, Dpto. de Sistemas Informáticos y Computación, Universidad Complutense de Madrid (June 2007)
21. Søndergaard, H., Sestoft, P.: Non-determinism in functional languages. The Computer Journal 35(5), 514–523 (1992)
22. Vado-Vírseda, R.d.: A demand-driven narrowing calculus with overlapping definitional trees. In: Proc. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP 2003), pp. 213–227. ACM Press, New York (2003)

Java Type Unification with Wildcards

Martin Plümicke

University of Cooperative Education Stuttgart/Horb
Florianstraße 15, D–72160 Horb
m.pluemicke@ba-horb.de

Abstract. With the introduction of Java 5.0 the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. The term

`Vector<? extends Vector<AbstractList<Integer>>>`

is for example a correct type in Java 5.0.

In this paper we present a type unification algorithm for Java 5.0 type terms. The algorithm unifies type terms, which are in subtype relationship. For this we define Java 5.0 type terms and its subtyping relation, formally.

As Java 5.0 allows wildcards as instances of generic types, the subtyping ordering contains infinite chains. We show that the type unification is still finitary. We give a type unification algorithm, which calculates the finite set of general unifiers.

1 Introduction

With the introduction of Java 5.0 [1] the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. For example the term

`Vector<? extends Vector<AbstractList<Integer>>>`

is a correct type in Java 5.0.

Considering all that, it is often rather difficult for a programmer to recognize whether such a complex type is the correct one for a given method or not.

This has caused us to develop a Java 5.0 type inference system which assists the programmer by calculating types automatically [2,3]. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principal types.

Following the ideas of [4], we reduce the Java 5.0 *type inference problem* to a Java 5.0 *type unification problem.* The Java 5.0 *type unification problem* is given as: For two type terms θ_1, θ_2 a substitution is demanded, such that $\sigma(\theta_1) \leq^* \sigma(\theta_2)$, where $\leq^*$ is the Java 5.0 subtyping relation. The type system of Java 5.0 is very similar to the type system of polymorphically order-sorted types, which is considered for the logical languages [5,6,7,8] and for the functional object–oriented language OBJ–P [9]. But in all approaches the type unification problem was not solved completely.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 223–240, 2009.

In [10] we have done a first step solving the type unification problem. We restricted the set of type terms, by disallowing wildcards, and presented a type unification algorithm for this approach.

In this paper we extend our algorithm to type terms with wildcards and prove its soundness and completeness. This means that we solve the original type unification problem and give a complete type unification algorithm. We will show, that the type unification problem is not still unitary, but finitary.

The paper is organized as follows. In the second section we formally describe the Java 5.0 type system including its inheritance hierarchy. In the third section we give an overview of the type unification problem. Then, we present the type unification algorithm, prove its soundness and completeness, and give an example. Finally, we close with a summary and an outlook.

2 Subtyping in Java 5.0

The Java 5.0 types are given as type terms over a type signature TS of class/ interface names and a set of bounded type variables BTV. While the type signature describes the arities of the class/interface names, a bound of a type variable restricts the allowed instantiated types to subtypes of the bound. For a type variable $\mathtt{a}$ bounded by the type ty we will write $\mathtt{a}|_{ty}$.

Example 1. Let the following Java 5.0 program be given:

```
class A<a> implements I<a> { ...}
class B<a> extends A<a> { ...}
class C<a extends I<b>,b> { ...}
interface I<a> { ...}
interface J<a> { ...}
class D<a extends B<a> & J<b>, b> { ...}
```

Then, the corresponding type signature TS is given as: $TS^{(\mathtt{a}|_{\mathtt{Object}})}=\{\, \mathtt{A}, \mathtt{B}, \mathtt{I}, \mathtt{J}\,\}$, $TS^{(\mathtt{a}|_{\mathtt{I<b>}}\ \mathtt{b}|_{\mathtt{Object}})} = \{\, \mathtt{C}\,\}$, and $TS^{(\mathtt{a}|_{\mathtt{B<a>\&J<b>}}\ \mathtt{b}|_{\mathtt{Object}})} = \{\, \mathtt{D}\,\}$.

As $\mathtt{A}, \mathtt{I} \in TS^{(\mathtt{a}|_{\mathtt{Object}})}$ and `Integer` is a subtype of `Object`, the terms `A<Integer>` and `I<Integer>` are Java 5.0 types. As `I<Integer>` is a subtype of itself and `A<Integer>` is also a subtype of `I<Integer>`, the terms `C<I<Integer>, Integer>` and `C<A<Integer>, Integer>` are also type terms. `J<Integer>` is no subtype of `I<Integer>` such that the term `C<J<Integer>, Integer>` is no Java 5.0 type.

For the definition of the inheritance hierarchy, the concept of Java 5.0 *simple types* and the concept of *capture conversion* is needed. For the definition of Java 5.0 simple types we refer to [1], Section 4.5, where Java 5.0 parameterized types are defined. We call them, in this paper, Java 5.0 *simple types* in contrast to the function types of methods. Java 5.0 simple types consists of the Java 5.0 types as in Example 1 presented and wildcard types like `Vector<? extends Integer>`. For the definition of the *capture conversion* we refer to [1] §5.1.10. The *capture conversion* transforms types with wildcard type arguments to equivalent types, where the wildcards are replaced by implicit type variables. The capture conversion of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ is denoted by $CC(\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}\,)$.

The inheritance hierarchy consists of two different relations: The "extends relation" (denoted by $<$) is explicitly defined in Java 5.0 programs by the *extends*, and the *implements* declarations, respectively. The "subtyping relation" (cp. [1], Section 4.10) is built as the reflexive, transitive, and instantiating closure of the extends relation.

In the following we will use ${}_?\theta$ as an abbreviation for the type term "`? extends` θ" and ${}^?\theta$ as an abbreviation for the type term "`? super` θ".

Definition 1 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(\, BTV\,)$). *Let TS be a type signature of a given Java 5.0 program and $<$ the corresponding extends relation. The* subtyping relation $\leq^*$ *is given as the reflexive and transitive closure of the smallest relation satisfying the following conditions:*

- *if $\theta < \theta'$ then $\theta \leq^* \theta'$.*
- *if $\theta_1 \leq^* \theta_2$ then $\sigma_1(\, \theta_1\,) \leq^* \sigma_2(\, \theta_2\,)$ for all substitutions $\sigma_1,\ \sigma_2 : BTV \to \mathsf{SType}_{TS}(\, BTV\,)$, where for each type variable a of θ_2 holds $\sigma_1(\, a\,) = \sigma_2(\, a\,)$ (soundness condition).*
- *$a|_{\theta_1 \& \ldots \& \theta_n} \leq^* \theta_i$ for $a \in BTV$ and $1 \leqslant i \leqslant n$*
- *It holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ if for each θ_i and θ'_i, respectively, one of the following conditions is valid:*
  - *$\theta_i = {}_?\overline{\theta}_i$, $\theta'_i = {}_?\overline{\theta}'_i$ and $\overline{\theta}_i \leq^* \overline{\theta}'_i$.*
  - *$\theta_i = {}^?\overline{\theta}_i$, $\theta'_i = {}^?\overline{\theta}'_i$ and $\overline{\theta}'_i \leq^* \overline{\theta}_i$.*
  - *$\theta_i, \theta'_i \in \mathsf{SType}_{TS}(\, BTV\,)$ and $\theta_i = \theta'_i$*
  - *$\theta'_i = {}_?\theta_i$*
  - *$\theta'_i = {}^?\theta_i$ (cp. [1] §4.5.1.1 type argument containment)*
- *Let $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>}$ be the capture conversions of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ and $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ then holds $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.*

It is surprising that the condition for σ_1 and σ_2 in the second item is not $\sigma_1(\, a\,) \leq^* \sigma_2(\, a\,)$, but $\sigma_1(\, a\,) = \sigma_2(\, a\,)$. This is necessary in order to get a sound type system. This property is the reason for the introduction of wildcards in Java 5.0 (cp. [1], §5.1.10).

The next example illustrates the subtyping definition.

Example 2. Let the Java 5.0 program from Example 1 be given again. Then the following relationships hold:

- `A<a>` $\leq^*$ `I<a>`, as `A<a>` $<$ `I<a>`
- `A<Integer>` $\leq^*$ `I<Integer>`, where $\sigma_1 = [a \mapsto \texttt{Integer}] = \sigma_2$
- `A<Integer>` $\leq^*$ `I<? extends Object>`, as `Integer` $\leq^*$ `Object`
- `A<Object>` $\leq^*$ `I<? super Integer>`, as `Integer` $\leq^*$ `Object`

There are elements of the extends relation, where the sub-terms of a type term are not variables, like `Matrix<a> extends Vector<Vector<a>>`. As elements like this must be handled especially during the unification (*adapt* rules, Fig. 2), we declare a further ordering on the set of type terms which we call the *finite closure* of the extends relation.

Definition 2 (Finite closure of $<$). *The* finite closure $\mathbf{FC}(\,<\,)$ *is the reflexive and transitive closure of pairs in the subtyping relation* $\leq^*$ *with* $C\texttt{<}a_1,\ldots,a_n\texttt{>} \leq^* D\texttt{<}\theta_1,\ldots,\theta_m\texttt{>}$*, where the* a_i *are type variables and the* θ_i *are type terms.*

If a set of bounded type variables BTV *is given, the finite closure* $\mathbf{FC}(\,<\,)$ *is extended to* $\mathbf{FC}(\,<, BTV\,)$*, by* $a|_\theta \leq^* a|_\theta$ *for* $a|_\theta \in BTV$*.*

Lemma 1. *The* finite closure $\mathbf{FC}(\,<\,)$ *is finite.*

Now we give a further example to illustrate the definition of the subtyping relation and the finite closure.

Example 3. Let the following Java 5.0 program be given.

```
abstract class AbstractList<a> implements List<a> {..}
class Vector<a> extends AbstractList<a> {..}
class Matrix<a> extends Vector<Vector<a>> {..}
```

Following the soundness condition of the Java 5.0 type system we get

$$\texttt{Vector<Vector<a>>} \not\leq^* \texttt{Vector<List<a>>},$$

but

$$\texttt{Vector<Vector<a>>} \leq^* \texttt{Vector<? extends List<a>>}.$$

The finite closure $\mathbf{FC}(\,<\,)$ is given as the reflexive and transitive closure of

$$\begin{aligned}\{\,&\texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}\\ &\texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>} \leq^* \texttt{AbstractList<Vector<a>>}\\ &\qquad\quad \leq^* \texttt{List<Vector<a>>}\,\}.\end{aligned}$$

Now we extend the set of simple types and the corresponding subtyping relation by wildcard types. Wildcard types cannot be used, explicitly, in Java 5.0 programs. But they are allowed as instances of type variables, which means that types like this occur implicitly during the type check of Java 5.0 programs (cf. Example 4).

Definition 3 (Extended simple types). *Let* $\mathsf{SType}_{TS}(\,BTV\,)$ *be a set of simple types. The corresponding set of* extended simple types *is given as*

$$\begin{aligned}\mathsf{ExtSType}_{TS}(\,BTV\,) = {} & \mathsf{SType}_{TS}(\,BTV\,) \\ & \cup \{\,?\,\} \\ & \cup \{\,?\,\mathsf{extends}\ \theta \mid \theta \in \mathsf{SType}_{TS}(\,BTV\,)\,\} \\ & \cup \{\,?\,\mathsf{super}\ \theta \mid \theta \in \mathsf{SType}_{TS}(\,BTV\,)\,\}\end{aligned}\quad.$$

According to this we extend the subtyping relation $\leq^*$ to *extended simple types.*

Definition 4 (Subtyping relation $\leq^*$ on $\mathsf{ExtSType}_{TS}(\,BTV\,)$). *Let* $\leq^*$ *be a subtyping relation on a given set of simple types* $\mathsf{SType}_{TS}(\,BTV\,)$*. Then* $\leq^*$ *is continued on the corresponding set of extended simple types* $\mathsf{ExtSType}_{TS}(\,BTV\,)$ *by: For* $\theta \leq^* \theta'$ *holds* $\theta \leq^* {}^{?}\theta'$*,* ${}_{?}\theta \leq^* \theta'$*, and* ${}_{?}\theta \leq^* {}^{?}\theta'$*.*

Example 4. Let us consider the class `Vector` with its methods `addElement` and `elementAt`, respectively and two classes `Sub` and `Super` with $\texttt{Sub} \leq^* \texttt{Super}$.

Let the following lines of Java 5.0 code be given:

```
Vector<? extends Super> v = new Vector<Sub> ();
Super sup = v.elementAt(0);
```

The type of the expression `v.elementAt(0)` is `? extends Super` and it holds $\texttt{? extends Super} \leq^* \texttt{Super}$.
Vice versa for

```
Vector<? super Super> v2 = new Vector<Super> ();
```

the methodcall `v2.addElement(new Sub());` is correct as $\texttt{Sub} \leq^* {}^{?}\texttt{Super}$. But

```
Super sup = v2.elementAt(0);   //not really correct
```

is not correct, as ${}^{?}\texttt{Super} \not\leq^* \texttt{Super}$.

Furthermore, the methodcall

```
v2.addElement(v.elementAt(0));
```

is correct, as ${}_{?}\texttt{Super} \leq^* {}^{?}\texttt{Super}$ holds.

3 Type Unification

In this section we consider the type unification problem of Java 5.0 type terms. The type unification problem is given as: For two type terms θ_1, θ_2 a substitution is demanded, such that

$$\sigma(\,\theta_1\,) \leq^* \sigma(\,\theta_2\,).$$

The algorithm solving the type unification problem is an important basis of the Java 5.0 type inference algorithm [3]. In the following we denote $\theta \lessdot \theta'$ for two type terms, which should be type unified.

3.1 Overview

First we give an overview of approaches considering the type unification problem on polymorphically order-sorted types. In [5] the type unification problem is mentioned as an open problem. For the logical language TEL in [5] an incomplete type inference algorithm is given. The incompleteness is caused by the fact, that subtype relationships of polymorphic types which have different arities (e.g. $\texttt{List(a)} < \texttt{myLi(a,b)}$) are allowed. This leads to the property that there are infinite chains in the type term ordering. In TEL for $\texttt{List(a)} \leq \texttt{myLi(a,b)}$ holds:

$$\texttt{List(a)} \leq \texttt{myLi(a,List(a))} \leq \texttt{myLi(a,myLi(a,List(a)))} \leq \ldots$$

Nevertheless, the type unification fails in some cases without infinite chains, although there is a unifier. Let for example $\texttt{nat} < \texttt{int}$, and the set of inequations

$\{\, \texttt{nat} \lessdot \texttt{a}, \texttt{int} \lessdot \texttt{a} \,\}$ be given, then $\{\, \texttt{a} \mapsto \texttt{nat} \,\}$ is determined, such that $\{\, \texttt{int} \lessdot \texttt{nat} \,\}$ fails, although $\{\, \texttt{a} \mapsto \texttt{int} \,\}$ is a unifier. For $\{\, \texttt{int} \lessdot \texttt{a}, \texttt{nat} \lessdot \texttt{a} \,\}$ the algorithm determines the correct unifier $\{\, \texttt{a} \mapsto \texttt{int} \,\}$.

In the typed logic programs of [7] subtype relationships of polymorphic types are allowed only between type constructors of the same arity. In this approach a *most general type unifier (mgtu)* is defined as an upper bound of different principal type unifiers. In general there are no upper bounds of two given type terms in the type term ordering, which means that there is in general no mgtu in the sense of [7]. For example for $\texttt{nat} < \texttt{int}$, $\texttt{neg} < \texttt{int}$, and the set of inequations $\{\, \texttt{nat} \lessdot \texttt{a}, \texttt{neg} \lessdot \texttt{a} \,\}$ the mgtu $\{\, \texttt{a} \mapsto \texttt{int} \,\}$ is determined. If the type term ordering is extended by $\texttt{int} < \texttt{index}$ and $\texttt{int} < \texttt{expr}$ (cp. Fig. 1), then there are three unifiers $\{\, \texttt{a} \mapsto \texttt{int} \,\}$, $\{\, \texttt{a} \mapsto \texttt{index} \,\}$, and $\{\, \texttt{a} \mapsto \texttt{expr} \,\}$, but none of them is a mgtu in the sense of [7].

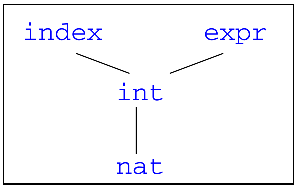


Fig. 1. Simple type ordering

The type system of PROTOS-L [8] was derived from TEL by disallowing any explicit subtype relationships between polymorphic type constructors.

In [8] a complete type unification algorithm is given, which can be extended to the type system of [7]. They solved the type unification problem for type term orderings following the restrictions of PROTOS-L respectively the restrictions of [7]. Additionally, the result of this paper is, that the type unification problem is not unitary, but finitary. This means in general that there is more than one general type unifier. For the above example the algorithm determines, where $\texttt{nat} < \texttt{int}$, $\texttt{neg} < \texttt{int}$, $\texttt{int} < \texttt{index}$, and $\texttt{int} < \texttt{expr}$ and the set of inequations $\{\, \texttt{nat} \lessdot \texttt{a}, \texttt{neg} \lessdot \texttt{a} \,\}$ is given, the three general unifiers $\{\, \texttt{a} \mapsto \texttt{int} \,\}$, $\{\, \texttt{a} \mapsto \texttt{index} \,\}$, and $\{\, \texttt{a} \mapsto \texttt{expr} \,\}$.

Finally, in [10] we disallowed wildcards in Java 5.0 type terms, which means that there is no subtyping in the arguments of the type term (soundness condition, Def. 1), but subtype relationship of type constructors with different arities is allowed. For type term orderings following this restriction we presented a type unification algorithm and proved that the type unification problem is also finitary. For example for $\texttt{myLi<b,a>} < \texttt{List<a>}$ and the set of inequations $\{\, \texttt{myLi<Integer,a>} \lessdot \texttt{List<Boolean>} \,\}$ the general unifier $\{\, \texttt{a} \mapsto \texttt{Boolean} \,\}$ is determined. For $\{\, \texttt{myLi<Integer,Integer>} \lessdot \texttt{List<Number>} \,\}$ the algorithm fails, as `Integer` is indeed a subtype of `Number`, but subtyping in the arguments is prohibited.

The type systems of TEL, respectively the other logical languages, and of Java 5.0 are very similar. The Java 5.0 type system has the same properties considering type unification, if it is restricted to simple types without parameter bounds (but including the wildcard constructions). The only difference is, that in TEL the number of arguments of a supertype type can be greater, whereas in Java 5.0 the number of arguments of a subtype can be greater. This means that infinite chains have a lower bound in TEL and an upper bound in Java 5.0. Let us consider again the following example: In TEL for `List(a)` $\leq$ `myLi(a,b)` holds:

$$\texttt{List(a)} \leq \texttt{myLi(a,List(a))} \leq \texttt{myLi(a,myLi(a,List(a)))} \leq \ldots$$

In contrast in Java 5.0 for `myLi<b, a>` $<$ `List<a>` holds:

$$\ldots \leq^* \texttt{myLi<}_?\texttt{myLi<}_?\texttt{List<a>, a>, a>} \leq^* \texttt{myLi<}_?\texttt{List<a>, a>} \leq^* \texttt{List<a>}$$

The open type unification problem of [5] is caused by these infinite chains. We will now present a solution for the open problem.

Our type unification algorithm bases on the algorithm by A. Martelli and U. Montanari [11] solving the original untyped unification problem. The main difference is, that in the original unification a unifier σ is demanded, such that $\sigma(\,\theta_1\,) = \sigma(\,\theta_2\,)$, whereas in our case a unifier σ is demanded, such that $\sigma(\,\theta_1\,) \leq^* \sigma(\,\theta_2\,)$. This means, that in the original case a result of the algorithm $a = \theta$ leads to one unifier $[a \mapsto \theta]$. In contrast to that, a result $a \leq^* \theta$ leads to a set of unifiers $\{\, [a \mapsto \overline{\theta}] \mid \overline{\theta} \leq^* \theta \,\}$ in our algorithm.

3.2 Type Unification Algorithm

In the description of the algorithm we denote also $\theta \lessdot \theta'$ for two type terms, which should be type unified. During the unification algorithm $\lessdot$ is replaced by $\lessdot_?$ and $\doteq$, respectively. $\theta \lessdot_? \theta'$ means that the two sub-terms θ and θ' of type terms should be unified, such that $\sigma(\,\theta\,)$ is a subtype of $\sigma(\,\theta'\,)$. $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\,\theta\,) = \sigma(\,\theta'\,)$.

The next definition gives a form of the set of equations for the end configuration of the algorithms.

Definition 5. (**Solved form**) A set of equations Eq is in *solved form*, if

$$Eq = \{\, a_1 \doteq \theta_1, \ldots, a_n \doteq \theta_n \,\}$$

where $a_1, \ldots, a_n \in BTV$ are pairwise different bounded type variables, $\theta_1, \ldots, \theta_n \in \mathsf{ExtSType}_{TS}(\,BTV\,)$ are extended simple types, and for all $i, j \in \{\, 1, \ldots, n \,\}$ holds: a_i does not occur in θ_j.

In the following, the type unification algorithm is shown. The type unification algorithm is given as follows

Input: Set of equations $Eq = \{\, \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \,\}$
Precondition: $\theta_i, \theta'_i \in \mathsf{SType}_{TS}(\,BTV\,)$ for $1 \leqslant i \leqslant n$.
Output: Set of all general type unifiers $Uni = \{\, \sigma_1, \ldots, \sigma_m \,\}$
Postcondition: For all $1 \leqslant j \leqslant m$ and for all $1 \leqslant i \leqslant n$ holds $(\sigma_j(\,\theta_i\,) \leq^* \sigma_j(\,\theta'_i\,))$.

$$\text{(adapt)}\quad \frac{Eq \cup \{\, D\text{<}\theta_1,\ldots,\theta_n\text{>} \lessdot D'\text{<}\theta'_1,\ldots,\theta'_m\text{>} \,\}}{Eq \cup \{\, D'\text{<}\overline{\theta}'_1,\ldots,\overline{\theta}'_m\text{>}[a_i \mapsto CC(\,\theta_i\,) \mid 1 \leqslant i \leqslant n] \lessdot D'\text{<}\theta'_1,\ldots,\theta'_m\text{>} \,\}}$$

where there are $\overline{\theta}'_1,\ldots,\overline{\theta}'_m$ with

– $(D\text{<}a_1,\ldots,a_n\text{>} \leq^* D'\text{<}\overline{\theta}'_1,\ldots,\overline{\theta}'_m\text{>}) \in \mathbf{FC}(\,<\,)$

$$\text{(adaptExt)}\quad \frac{Eq \cup \{\, D\text{<}\theta_1,\ldots,\theta_n\text{>} \lessdot_{?}\, {}_{?}D'\text{<}\theta'_1,\ldots,\theta'_m\text{>} \,\}}{Eq \cup \{\, D'\text{<}\overline{\theta}'_1,\ldots,\overline{\theta}'_m\text{>}[a_i \mapsto CC(\,\theta_i\,) \mid 1 \leqslant i \leqslant n] \lessdot_{?}\, {}_{?}D'\text{<}\theta'_1,\ldots,\theta'_m\text{>} \,\}}$$

where $D \in \Theta^{(n)}$ or $D = {}_{?}\overline{X}$ with $\overline{X} \in \Theta^{(n)}$ and
there are $\overline{\theta}'_1,\ldots,\overline{\theta}'_m$ with

– ${}_{?}D'\text{<}\overline{\theta}'_1,\ldots,\overline{\theta}'_m\text{>}) \in \mathbf{grArg}(\, D\text{<}a_1,\ldots,a_n\text{>}\,)$

$$\text{(adaptSup)}\quad \frac{Eq \cup \{\, D'\text{<}\theta'_1,\ldots,\theta'_m\text{>} \lessdot_{?}\, {}^{?}D\text{<}\theta_1,\ldots,\theta_n\text{>} \,\}}{Eq \cup \{\, D'\text{<}\overline{\theta}'_1,\ldots,\overline{\theta}'_m\text{>}[a_i \mapsto CC(\,\theta_i\,) \mid 1 \leqslant i \leqslant n] \lessdot_{?}\, {}_{?}D'\text{<}\theta'_1,\ldots,\theta'_m\text{>} \,\}}$$

where $D' \in \Theta^{(n)}$ or $D' = {}^{?}\overline{X}$ with $\overline{X} \in \Theta^{(n)}$ and
there are $\overline{\theta}'_1,\ldots,\overline{\theta}'_m$ with

– ${}^{?}D\text{<}a_1,\ldots,a_n\text{>} \in \mathbf{grArg}(\, D'\text{<}\overline{\theta}'_1,\ldots,\overline{\theta}'_m\text{>}\,)$

Fig. 2. Java 5.0 type unification adapt rules

The algorithm itself is given in seven steps:

1. Repeated application of the *adapt* rules (Fig. 2), the *reduce* rules, the *erase* rules and the *swap* rule (Fig. 3) to all elements of Eq. The end configuration of Eq is reached if for each element no rule is applicable.
2. Eq'_1 = *Subset of pairs, where both type terms are type variables*
3. $Eq'_2 = Eq \backslash Eq'_1$
4. Eq'_{set}

$$
\begin{aligned}
= \{\, Eq'_1 \,\} \times \Big(\bigotimes_{(a \lessdot \theta') \in Eq'_2} \{\, ([a \doteq \theta] \cup \sigma) \mid\; & (\overline{\theta} \leq^* C\text{<}\theta_1,\ldots,\theta_n\text{>}) \in \mathbf{FC}(\,<\,), \\
& \overline{\theta}' \in \{\, C\text{<}\theta'_1,\ldots,\theta'_n\text{>} \\
& \qquad \mid \theta'_i \in \mathbf{grArg}(\,\theta_i\,), 1 \leqslant i \leqslant n \,\} \\
& \sigma \in \mathsf{Unify}(\,\overline{\theta}', \theta'\,), \\
& \theta \in \mathbf{smaller}(\,\sigma(\,\overline{\theta}\,)\,) \,\}\Big) \\
\times \Big(\bigotimes_{(a \lessdot_{?}\, {}_{?}\theta') \in Eq'_2} \{\, ([a \doteq \theta] \cup \sigma) \mid\; & (\overline{\theta} \leq^* C\text{<}\theta_1,\ldots,\theta_n\text{>}) \in \mathbf{FC}(\,<\,), \\
& \overline{\theta}' \in \{\, C\text{<}\theta'_1,\ldots,\theta'_n\text{>} \\
& \qquad \mid \theta'_i \in \mathbf{grArg}(\,\theta_i\,), 1 \leqslant i \leqslant n \,\} \\
& \sigma \in \mathsf{Unify}(\,\overline{\theta}', \theta'\,), \\
& \theta \in \mathbf{smArg}(\,\sigma(\,{}_{?}\overline{\theta}\,)\,) \,\}\Big) \\
\times \Big(\bigotimes_{(a \lessdot_{?}\, {}^{?}\theta') \in Eq'_2} \{\, [a \doteq \theta'] \mid\; & \theta' \in \mathbf{smArg}(\,{}^{?}\theta'\,) \,\}\Big) \\
\times \Big(\bigotimes_{(a \lessdot_{?}\, \theta') \in Eq'_2} \{\, [a \doteq \theta'] \,\}\Big) &
\end{aligned}
$$

$$\text{(reduceUp)}\ \frac{Eq \cup \{\, \theta \lessdot {}^{?}\theta' \,\}}{Eq \cup \{\, \theta \lessdot \theta' \,\}} \qquad \text{(reduceUpLow)}\ \frac{Eq \cup \{\, {}_{?}\theta \lessdot {}^{?}\theta' \,\}}{Eq \cup \{\, \theta \lessdot \theta' \,\}}$$

$$\text{(reduceLow)}\ \frac{Eq \cup \{\, {}_{?}\theta \lessdot \theta' \,\}}{Eq \cup \{\, \theta \lessdot \theta' \,\}}$$

$$\text{(reduce1)}\ \frac{Eq \cup \{\, C<\theta_1, \ldots, \theta_n> \lessdot D<\theta'_1, \ldots, \theta'_n> \,\}}{Eq \cup \{\, \theta_{\pi(1)} \lessdot_{?} \theta'_1, \ldots, \theta_{\pi(n)} \lessdot_{?} \theta'_n \,\}}$$

where

- $C<a_1, \ldots, a_n> \leq^* D<a_{\pi(1)}, \ldots, a_{\pi(n)}>$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- π is a permutation

$$\text{(reduceExt)}\ \frac{Eq \cup \{\, X<\theta_1, \ldots, \theta_n> \lessdot_{?} {}_{?}Y<\theta'_1, \ldots, \theta'_n> \,\}}{Eq \cup \{\, \theta_{\pi(1)} \lessdot_{?} \theta'_1, \ldots, \theta_{\pi(n)} \lessdot_{?} \theta'_n \,\}}$$

where

- ${}_{?}Y<a_{\pi(1)}, \ldots, a_{\pi(n)}> \in \mathbf{grArg}(\, X<a_1, \ldots, a_n> \,)$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- π is a permutation
- $X \in \varTheta^{(n)}$ or $X = {}_{?}\overline{X}$ with $\overline{X} \in \varTheta^{(n)}$.

$$\text{(reduceSup)}\ \frac{Eq \cup \{\, X<\theta_1, \ldots, \theta_n> \lessdot_{?} {}^{?}Y<\theta'_1, \ldots, \theta'_n> \,\}}{Eq \cup \{\, \theta'_1 \lessdot_{?} \theta_{\pi(1)}, \ldots, \theta'_n \lessdot_{?} \theta_{\pi(n)} \,\}}$$

where

- ${}^{?}Y<a_{\pi(1)}, \ldots, a_{\pi(n)}> \in \mathbf{grArg}(\, X<a_1, \ldots, a_n> \,)$
- $\{\, a_1, \ldots, a_n \,\} \subseteq BTV$
- π is a permutation
- $X \in \varTheta^{(n)}$ or $X = {}^{?}\overline{X}$ with $\overline{X} \in \varTheta^{(n)}$.

$$\text{(reduceEq)}\ \frac{Eq \cup \{\, X<\theta_1, \ldots, \theta_n> \lessdot_{?} X<\theta'_1, \ldots, \theta'_n> \,\}}{Eq \cup \{\, \theta_{\pi(1)} \doteq \theta'_1, \ldots, \theta_{\pi(n)} \doteq \theta'_n \,\}}$$

$$\text{(reduce2)}\ \frac{Eq \cup \{\, C<\theta_1, \ldots, \theta_n> \doteq C<\theta'_1, \ldots, \theta'_n> \,\}}{Eq \cup \{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}}$$

where

- $C \in \varTheta^{(n)}$ or $C = {}_{?}\overline{C}$ or $C = {}^{?}\overline{C}$ with $\overline{C} \in \varTheta^{(n)}$, respectively.

$$\text{(erase1)}\ \frac{Eq \cup \{\, \theta \lessdot \theta' \,\}}{Eq}\ \theta \leq^* \theta' \qquad \text{(erase2)}\ \frac{Eq \cup \{\, \theta \lessdot_{?} \theta' \,\}}{Eq}\ \theta' \in \mathbf{grArg}(\, \theta \,)$$

$$\text{(erase3)}\ \frac{Eq \cup \{\, \theta \doteq \theta' \,\}}{Eq}\ \theta = \theta' \qquad \text{(swap)}\ \frac{Eq \cup \{\, \theta \doteq a \,\}}{Eq \cup \{\, a \doteq \theta \,\}}\ \theta \notin BTV,\ a \in BTV$$

Fig. 3. Java 5.0 type unification rules with wildcards

$$
\begin{array}{ll}
\times\, (\displaystyle\bigotimes_{(\theta \lessdot a) \in Eq'_2} & \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{greater}(\,\theta\,)\, \}) \\
\times\, (\displaystyle\bigotimes_{({}_{?}\theta \,\lessdot_{?}\, a) \in Eq'_2} & \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{grArg}(\,{}_{?}\theta\,)\, \}) \\
\times\, (\displaystyle\bigotimes_{({}^{?}\theta \,\lessdot_{?}\, a) \in Eq'_2} & \{\, ([a \doteq {}^{?}\theta'] \cup \sigma) \mid \begin{array}[t]{l} (\overline{\theta} \leq^* C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}) \in \mathbf{FC}(\, < \,), \\ \overline{\theta}' \in \{\, C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \\ \qquad\qquad \mid \theta'_i \in \mathbf{grArg}(\,\theta_i\,), 1 \leqslant i \leqslant n \,\} \\ \sigma \in \mathsf{Unify}(\,\overline{\theta}', \theta\,), \\ \theta' \in \mathbf{smaller}(\,\sigma(\,\overline{\theta}\,)\,)\, \}) \end{array} \\
\times\, (\displaystyle\bigotimes_{(\theta \,\lessdot_{?}\, a) \in Eq'_2} & \{\, [a \doteq \theta'] \mid \theta' \in \mathbf{grArg}(\,\theta\,)\, \}) \\
\times\, \{\, [a \doteq \theta \mid (a \doteq \theta) \in Eq'_2] \,\} &
\end{array}
$$

5. Application of the following *subst* rule

$$
(\text{subst})\ \frac{Eq'' \cup \{\, a \doteq \theta \,\}}{Eq''[a \mapsto \theta] \cup \{\, a \doteq \theta \,\}}\ a \text{ occurs in } Eq'' \text{ but not in } \theta
$$

for each $a \doteq \theta$ in each element of $Eq' \in Eq'_{set}$.

6. (a) Foreach $Eq' \in Eq'_{set}$ which has changed in the last step start again with the first step.
   (b) Build the union Eq''_{set} of all results of (a) and all $Eq' \in Eq'_{set}$ which has not changed in the last step.
7. $\mathit{Uni} = \{\, \sigma \mid Eq'' \in Eq''_{set}, Eq'' \text{ is in solved form},$
   $\sigma = \{\, a \mapsto \theta \mid (a \doteq \theta) \in Eq'' \,\}\,\}$

In the algorithm the unbounded wildcard "?" is denoted as the equivalent bounded wildcard "`? extends Object`".

Furthermore, there are functions **greater** and **grArg**, respectively **smaller** and **smArg**. These functions determine all supertypes, respectively all subtypes by pattern matching with the elements of the finite closure. The functions **greater** and **smaller** determine the supertypes respectively subtypes of simple types, while **grArg** and **smArg** determine the supertypes respectively the subtypes of sub-terms, which are allowed as arguments in type terms.

The function Unify is the ordinary unification.

Now we will explain the rules in Fig. 2 and 3.

adapt rules: The *adapt* rules adapt type term pairs, which are built by class declarations like

`class` C`<`$a_1, \ldots, a_n$`>` `extends` D`<`D_1`<`$\ldots$`>`$, \ldots, D_m$`<`$\ldots$`>>`.

The smaller type is replaced by a type term, which has the same outermost type name as the greater type. Its sub-terms are determined by the finite closure. The instantiations are maintained.

The *adaptExt* and *adaptSup* rule are the corresponding rules to the *adapt* rule for sub-terms of type terms.

- **reduce rules:** The rules *reduceUp*, *reduceUpLow*, and *reduceLow* correspond to the extension of the subtyping ordering on extended simple types (Def. 4).

  The *reduce1* rule follows from the construction of the subtyping relation $\leq^*$, where $C(\,a_1, \ldots, a_n\,) \ < \ D(\,a_{\pi(\,1\,)}, \ldots, a_{\pi(\,n\,)}\,)$ leads to $C(\,\theta_1, \ldots, \theta_n\,) \leq^* \ D(\,\theta'_1, \ldots, \theta'_n\,)$ if and only if $\theta'_i \in \mathbf{grArg}(\,\theta_{\pi(\,i\,)}\,)$ for $1 \leqslant i \leqslant n$.

  The *reduceExt* and the *reduceSup* rules are the corresponding rules to the *reduce1* rule for sub-terms of type terms.

  The *reduceEq* and the *reduce2* rule ensures, that sub-terms must be equal, if there are no wildcards (soundness condition of the Java 5.0 type system, Def. 1).
- **erase rules:** The *erase* rules erase type term pairs, which are in the respective relationship.
- **swap rule:** The *swap* rule swaps type term pairs, such that type variables are mapped to type terms, not vice versa.

Now we give an example for the type unification algorithm.

Example 5. In this example we use the standard Java 5.0 types `Number`, `Integer`, `Stack`, `Vector`, `AbstractList`, and `List`. It holds `Integer` < `Number` and `Stack<a>` < `Vector<a>` < `AbstractList<a>` < `List<a>`.

As a start configuration we use
$\{\,(\texttt{Stack<a>} \lessdot \texttt{Vector<}{}_?\texttt{Number>}),\ (\texttt{AbstractList<Integer>} \lessdot \texttt{List<a>})\,\}$.

In the first step the reduce1 rule is applied twice:

$$\{\,\texttt{a} \lessdot_? {}_?\texttt{Number},\ \texttt{Integer} \lessdot_? \texttt{a}\,\}$$

With the second and the third step we receive in step four:

$$\begin{array}{l}
\{\ \{\,\texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number}\,\},\\
\quad \{\,\texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer}\,\},\\
\quad \{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number}\,\},\\
\quad \{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer}\,\},\\
\quad \{\,\texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number}\,\},\\
\quad \{\,\texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer}\,\}, \{\,\texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer}\,\},\\
\quad \{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number}\,\},\\
\quad \{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer}\,\}\{\,\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer}\,\}\ \}
\end{array}$$

In the fifth step the rule *subst* is applied:

$$\begin{array}{l}
\{\,\{\,\texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \underline{\{\,{}_?\texttt{Number} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number}\,\}},\\
\ \{\,{}_?\texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer}\,\}, \{\,{}^?\texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer}\},\\
\ \{\,\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,{}_?\texttt{Number} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number}\,\},\\
\ \{\,{}_?\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer}\,\}, \{\,{}^?\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer}\,\},\\
\ \{\,\texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}, \{\,{}_?\texttt{Number} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number}\,\},\\
\ \underline{\{{}_?\texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer}\}}, \{{}^?\texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer}\},\\
\ \underline{\{\,\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\,\}}, \{\,{}_?\texttt{Number} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number}\,\},\\
\ \{{}_?\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer}\}\{{}^?\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer}\}\ \}
\end{array}$$

The underlined sets of type term pairs lead to unifiers.

Now we have to continue with the first step (step 6(a)). With the application of the *erase3* rule and step 7, we get three general type unifiers:

$$\{\, \{\, a \mapsto {}_?\mathtt{Number} \,\}, \{\, a \mapsto {}_?\mathtt{Integer} \,\}, \{\, a \mapsto \mathtt{Integer} \,\} \,\}.$$

The following example shows, how the algorithm works for subtype relationships with different numbers of arguments and wildcards, which causes infinite chains (cp. Section 3.1).

Example 6. Let $\mathtt{myLi{<}b,a{>}} < \mathtt{List{<}a{>}}$ and the start configuration $\{\, \mathtt{List{<}x{>}} \lessdot \mathtt{List{<}{}_?List{<}Integer{>}{>}} \,\}$ be given. In the first step the reduce 1 rule is applied: $\{\, \mathtt{x} \lessdot_? {}_?\mathtt{List{<}Integer{>}} \,\}$. With the fourth step we get the result:

$$\begin{aligned} \{\, &\{\, \mathtt{x} \mapsto \mathtt{List{<}Integer{>}} \,\}, \{\, \mathtt{x} \mapsto {}_?\mathtt{List{<}Integer{>}} \,\}, \\ &\{\, \mathtt{x} \mapsto \mathtt{myLi{<}b,Integer{>}} \,\}, \{\, \mathtt{x} \mapsto {}_?\mathtt{myLi{<}b,Integer{>}} \,\} \,\}. \end{aligned}$$

All other infinite numbers of unifiers are instances of these general unifiers (e.g. $\{\, \mathtt{x} \mapsto {}_?\mathtt{myLi{<}{}_?List{<}Integer{>},Integer{>}} \,\}$ or $\{\, \mathtt{x} \mapsto {}_?\mathtt{myLi{<}{}_?myList{<}b,Integer{>},Integer{>}} \,\}$).

The following theorem shows, that the type unification problem is solved by the type unification algorithm.

Theorem 1. *The type unification algorithm determines exactly all general type unifiers for a given set of type term pairs. This means that the algorithm is sound and complete.*

Proof. We do the proof in two steps. First we prove the *soundness* and then the *completeness.*

Soundness

We have to prove that each calculated result of the algorithm is a general unifier of the corresponding input. We do the proof as follows: For an arbitrary result σ we show that if σ is a general unifier of the set of type terms after a transformation, then σ is also a general unifier of the set of type term pairs before the transformation.

Let $\sigma = \{\, a_1 \mapsto \theta_1, \ldots, a_n \mapsto \theta_n \,\}$ be a result of the algorithm.

Step seven: In step seven the equations $a_i \doteq \theta_i$ are transformed to the mappings $a_i \mapsto \theta_i$. It is obvious that σ is a general unifier of

$$\{\, a_1 \doteq \theta_1, \ldots, a_n \doteq \theta_n \,\}.$$

Step six: In step six different sets of type term pairs are united. This means that σ is also a general unifier before step six is applied.

Step five: The type variables a are subsituted by θ. As for $a \doteq \theta$ holds $\sigma(\, a\,) = \sigma(\, \theta\,)$, σ is also a general unifier of the set of type terms before the substitution.

Step four: We have to consider eight different cases in step four.

$\underline{(a \lessdot \theta') \in Eq'_2}$**:** In this case type term pairs of the form $(a \lessdot \theta')$ are transformed to $[(a \doteq \theta) \cup \sigma']$ for a type term θ with $\theta \in \mathbf{smaller}(\, \sigma'(\, \theta'\,)\,)$. If σ is a general unifier of $[(a \doteq \theta) \cup \sigma']$ then σ is also a general unifier of $(a \lessdot \theta')$ for $\theta \in \mathbf{smaller}(\, \sigma'(\, \theta'\,)\,)$.

$\underline{(a \lessdot_? {}_?\theta') \in Eq'_2}$: This case is analogous to the first case.

$\underline{(a \lessdot_? {}^?\theta') \in Eq'_2}$: In this case type term pairs of the form $(a \lessdot_? {}^?\theta')$ are transformed to $a \doteq \theta'$ for a type term θ' with $\theta' \in \mathbf{smArg}(\,{}^?\theta'\,)$.

If σ is a general unifier of $(a \doteq \theta')$ then σ is also a general unifier of $(a \lessdot_? {}^?\theta')$ for $\theta' \in \mathbf{smArg}(\,{}^?\theta'\,)$.

$\underline{(a \lessdot_? \theta') \in Eq'_2}$: It is obvious, that a general unifier of $(a \doteq \theta')$ is also a general unifier of $(a \lessdot_? \theta')$.

$\underline{(\theta \lessdot a) \in Eq'_2}$, $\underline{({}_?\theta \lessdot_? a) \in Eq'_2}$: The proof of these cases is analogous to the proof of case $(a \lessdot_? {}^?\theta') \in Eq'_2$

$\underline{({}^?\theta \lessdot_? a) \in Eq'_2}$: The proof of this case is analogous to the proof of case $(a \lessdot \theta') \in Eq'_2$.

$\underline{(\theta \lessdot_? a) \in Eq'_2}$: The proof of this case is analogous to the proof of case $(a \lessdot_? {}^?\theta') \in Eq'_2$

Step two and three: In these steps the pairs are only filtered. This means that the pairs are unchanged. But this means that σ is also a general unifier before applying the steps.

Step one: The soundness of the *reduce2* rule, the *erase3* rule, and the *swap* rule follows directly from the original unification algorithm of [11] as they are unchanged.

The soundness of the *erase1* rule and the *erase2* rule are obvious.

The soundness of the *reduceUp* rule, the *reduceUpLow* rule, and the *reduceLow* rule follows directly from the definition of the subtyping ordering on $\mathsf{ExtSType}_{TS}(\,BTV\,)$ (Def. 4), as the following relationships are equivalent $\theta \leq^* \theta'$, ${}_?\theta \leq^* \theta'$, $\theta \leq^* {}^?\theta'$, and ${}_?\theta \leq^* {}^?\theta'$.

***reduce1* rule:** If σ is a general unifier of $\theta_{\pi(\,i\,)} \lessdot_? \theta'_i$ for $1 \leqslant i \leqslant n$, it holds $\sigma(\,\theta'_i\,) \in \mathbf{grArg}(\,\sigma(\,\theta_{\pi(\,i\,)}\,)\,)$ for $1 \leqslant i \leqslant n$. With

$$C\texttt{<}a_1, \ldots, a_n\texttt{>} \leq^* D\texttt{<}a_{\pi(\,1\,)}, \ldots, a_{\pi(\,n\,)}\texttt{>}$$

follows also by Def. 1:

$$\sigma(\,C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}\,) \leq^* \sigma(\,D\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}\,)$$

***reduceExt* rule:** If σ is a general unifier of $\theta_{\pi(\,i\,)} \lessdot_? \theta'_i$ for $1 \leqslant i \leqslant n$, it holds $\sigma(\,\theta'_i\,) \in \mathbf{grArg}(\,\sigma(\,\theta_{\pi(\,i\,)}\,)\,)$ for $1 \leqslant i \leqslant n$. With

$${}_?Y\texttt{<}a_{\pi(\,1\,)}, \ldots, a_{\pi(\,n\,)}\texttt{>} \in \mathbf{grArg}(\,X\texttt{<}a_1, \ldots, a_n\texttt{>}\,)$$

follows also by Def. 1:

$$\sigma(\,{}_?Y\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}\,) \in \mathbf{grArg}(\,\sigma(\,X\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}\,)\,),$$

which means that σ is a general of $X\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot_? {}_?Y\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.

***reduceSup* rule:** If σ is a general unifier of $\theta'_i \lessdot_? \theta_{\pi(i)}$ for $1 \leqslant i \leqslant n$, it holds $\sigma(\theta_{\pi(i)}) \in \mathbf{grArg}(\sigma(\theta'_i))$ for $1 \leqslant i \leqslant n$. With

$${}^{?}Y<a_{\pi(1)},\ldots,a_{\pi(n)}> \in \mathbf{grArg}(X<a_1,\ldots,a_n>)$$

follows also by Def. 1:

$$\sigma({}^{?}Y<\theta'_1,\ldots,\theta'_n>) \in \mathbf{grArg}(\sigma(X<\theta_1,\ldots,\theta_n>)),$$

which means that σ is a general unifier of $X<\theta_1,\ldots,\theta_n> \lessdot_? {}^{?}Y<\theta'_1,\ldots,\theta'_n>$.

***reduceEq* rule:** If it holds for the general unifier σ: $\sigma(\theta_{\pi(i)}) = \sigma(\theta'_i)$ for $1 \leqslant i \leqslant n$ then follows by Def. 1:

$$\sigma(X<\theta'_1,\ldots,\theta'_n>) \in \mathbf{grArg}(\sigma(X<\theta_1,\ldots,\theta_n>)),$$

which means that σ is a general unifier of $X<\theta_1,\ldots,\theta_n> \lessdot_? X<\theta'_1,\ldots,\theta'_n>$.

***adapt* rule:** If it holds for the general unifier σ:

$$\sigma(D'<\overline{\theta}'_1,\ldots,\overline{\theta}'_m>[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n]) \leq^* \sigma(D'<\theta'_1,\ldots,\theta'_m>),$$

and

$$D<a_1,\ldots,a_n> \leq^* D'<\overline{\theta}'_1,\ldots,\overline{\theta}'_m>$$

then follows also by Def. 1:

$$\sigma(D<\theta_1,\ldots,\theta_n>) \leq^* \sigma(D'<\theta'_1,\ldots,\theta'_m>).$$

***adaptExt* rule:** If σ is a general unifier of $D'<\overline{\theta}'_1,\ldots,\overline{\theta}'_m>[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n] \lessdot_? {}_{?}D'<\theta'_1,\ldots,\theta'_m>$, it holds:

$$\begin{aligned}&\sigma({}_{?}D'<\theta'_1,\ldots,\theta'_m>)\\ &\qquad \in \mathbf{grArg}(\sigma(D'<\overline{\theta}'_1,\ldots,\overline{\theta}'_m>[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n])).\end{aligned}$$

With

$${}_{?}D'<\overline{\theta}'_1,\ldots,\overline{\theta}'_m> \in \mathbf{grArg}(D<a_1,\ldots,a_n>)$$

follows also by Def. 1:

$$\sigma({}_{?}D'<\theta'_1,\ldots,\theta'_m>) \in \mathbf{grArg}(\sigma(D<\theta_1,\ldots,\theta_n>)),$$

which means that σ is a general unifier of $D<\theta_1,\ldots,\theta_n> \lessdot_? {}_{?}D'<\theta'_1,\ldots,\theta'_m>$.

***adaptSup* rule:** If σ is a general unifier of $D'<\overline{\theta}'_1,\ldots,\overline{\theta}'_m>[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n] \lessdot_? {}_{?}D'<\theta'_1,\ldots,\theta'_m>$, it holds:

$$\begin{aligned}&\sigma({}_{?}D'<\theta'_1,\ldots,\theta'_m>)\\ &\qquad \in \mathbf{grArg}(\sigma(D'<\overline{\theta}'_1,\ldots,\overline{\theta}'_m>[a_i \mapsto CC(\theta_i) \mid 1 \leqslant i \leqslant n])).\end{aligned}$$

With

$${}^{?}D<a_1,\ldots,a_n> \in \mathbf{grArg}(D'<\overline{\theta}'_1,\ldots,\overline{\theta}'_m>)$$

follows also by Def. 1:

$$\sigma({}^{?}D<\theta_1,\ldots,\theta_n>) \in \mathbf{grArg}(\sigma(D'<\theta'_1,\ldots,\theta'_m>)),$$

which means that σ is a general unifier of $D'<\theta'_1,\ldots,\theta'_m> \lessdot_? {}^{?}D<\theta_1,\ldots,\theta_n>$.

Summarized, this means that σ is also a general unifier of the type term set before the application of this step.

We have proved for all transformations that if a substitution is a general unifier of the result the same substitution is also a general unifier of the input. This means that the algorithm is sound.

Completeness

For the completeness we have to prove for an arbitrary general unifier σ of an input set of type term pairs, that σ is determined by the algorithm. We do the proof, as we show for each transformation of the algorithm, if σ is a general unifier of the set of type terms before a transformation is done, σ is also a general unifier of at least one set of type term pairs after the transformation.

Let σ be a general unifier of $Eq = \{\, \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \,\}$

Step one: For the *reduce2* rule, the *erase3* rule, and the *swap* rule it follows directly from the original unification algorithm in [11] that all general unifier are obtained, as the rules are unchanged.

It is obvious that the *erase1* rule and the *erase2* rule obtain all unifiers.

For the *reduceUp* rule, the *reduceUpLow* rule, and the *reduceLow* rule it follows directly from the definition of the subtyping ordering on $\mathsf{ExtSType}_{TS}(\, BTV \,)$ (Def. 4), that they obtain all unifiers, as the following relationships are equivalent $\theta \leq^* \theta'$, ${}_{?}\theta \leq^* \theta'$, $\theta \leq^* {}^{?}\theta'$, and ${}_{?}\theta \leq^* {}^{?}\theta'$.

***reduce1* rule:** If σ is a general unifier of $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ from $C\texttt{<}a_1, \ldots, a_n\texttt{>} \leq^* D\texttt{<}a_{\pi(\,1\,)}, \ldots, a_{\pi(\,n\,)}\texttt{>}$ with Def. 1 follows $\sigma(\, \theta'_i \,) \in \mathbf{grArg}(\, \sigma(\, \theta_{\pi(\,i\,)} \,)\,)$ for $1 \leqslant i \leqslant n$. This means that σ is also a general unifier of $\theta_{\pi(\,i\,)} \lessdot_? \theta'_i$ for $1 \leqslant i \leqslant n$.

***reduceExt* rule:** If σ is a general unifier of $X\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot {}_{?}Y\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ from ${}_{?}Y\texttt{<}a_{\pi(\,1\,)}, \ldots, a_{\pi(\,n\,)}\texttt{>} \in \mathbf{grArg}(\, X\texttt{<}a_1, \ldots, a_n\texttt{>}\,)$ with Def. 1 follows $\sigma(\, \theta'_i \,) \in \mathbf{grArg}(\, \sigma(\, \theta_{\pi(\,i\,)} \,)\,)$ for $1 \leqslant i \leqslant n$. This means that σ is also a general unifier of $\theta_{\pi(\,i\,)} \lessdot_? \theta'_i$ for $1 \leqslant i \leqslant n$.

***reduceSup* rule:** If σ is a general unifier of $X\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot {}^{?}Y\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ from ${}^{?}Y\texttt{<}a_{\pi(\,1\,)}, \ldots, a_{\pi(\,n\,)}\texttt{>} \in \mathbf{grArg}(\, X\texttt{<}a_1, \ldots, a_n\texttt{>}\,)$ with Def. 1 follows $\sigma(\, \theta_{\pi(\,i\,)} \,) \in \mathbf{grArg}(\, \sigma(\, \theta'_i \,)\,)$ for $1 \leqslant i \leqslant n$. This means that σ is also a general unifier of $\theta'_i \lessdot_? \theta_{\pi(\,i\,)}$ for $1 \leqslant i \leqslant n$.

***reduceEq* rule:** If σ is a general unifier of $X\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot_? X\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ with Def. 1 follows $\sigma(\, \theta_i \,) = \sigma(\, \theta'_i \,)$ for $1 \leqslant i \leqslant n$. This means that σ is also a general unifier of $\theta_i \doteq \theta'_i$ for $1 \leqslant i \leqslant n$.

***adapt* rule:** If σ is a general unifier of $D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>}$ from $D\texttt{<}a_1, \ldots, a_n\texttt{>} \leq^* D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>} \in \mathbf{FC}(\, \leq^* \,)$ with Def. 1 follows that σ is also a general unifier of

$$D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>}[a_i \mapsto CC(\, \theta_i \,) \mid 1 \leqslant i \leqslant n] \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>}.$$

***adaptExt* rule:** If σ is a general unifier of $D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot_? {}_{?}D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>}$ from ${}_{?}D'\texttt{<}\overline{\theta}'_1, \ldots, \overline{\theta}'_m\texttt{>} \in \mathbf{grArg}(\, D\texttt{<}a_1, \ldots, a_n\texttt{>}\,)$ follows that σ is also a general unifier of

$$D'<\overline{\theta}'_1, \ldots, \overline{\theta}'_m>[a_i \mapsto CC(\, \theta_i\,) \mid 1 \leqslant i \leqslant n] \lessdot_{?}{}_{?}D'<\theta'_1, \ldots, \theta'_m>.$$

***adaptSup* rule:** If σ is a general unifier of $D'<\theta'_1, \ldots ,\theta'_m> \lessdot_{?}{}^{?}D<\theta_1, \ldots ,\theta_m>$ from ${}^{?}D<a_1, \ldots, a_n> \in \mathbf{grArg}(\, D'<\overline{\theta}'_1, \ldots, \overline{\theta}'_m>\,)$ follows that σ is also a general unifier of

$$D'<\overline{\theta}'_1, \ldots, \overline{\theta}'_m>[a_i \mapsto CC(\, \theta_i\,) \mid 1 \leqslant i \leqslant n] \lessdot_{?}{}_{?}D'<\theta'_1, \ldots, \theta'_m>.$$

This means that after step one σ is still a general unifier of the transformed set of type term pairs Eq.

Step two and three: In these steps the pairs are only filtered. This means that the pairs are unchanged. But this means that σ is also a general unifier after applying the steps.

Step four: We have to consider eight different cases:

$\underline{(a \lessdot \theta') \in Eq'_2}$**:** It holds $\sigma(\, a\,) \leq^* \sigma(\, \theta'\,)$. With Def. 1 follows

- there is a $(\overline{\theta} \leq^* C<\theta'_1, \ldots, \theta'_n>) \in \mathbf{FC}(\, < \,)$
- for $1 \leqslant i \leqslant n$: there are $(\theta''_i \in \mathbf{grArg}(\, \theta'_i\,))$,
- there is a substitution $\sigma' = \mathsf{Unify}(\, C<\theta''_1, \ldots, \theta''_n>, \theta'\,)$, and $\theta \in \mathbf{smaller}(\, \sigma'(\, \overline{\theta}\,)\,)$ such that $\sigma(\, a\,) = \theta$.

This means for the pair $(a \lessdot \theta') \in Eq'_2$, where σ is a general unifier, there is a set of equations in Eq_{set} after the transformation with a pair $(a \doteq \theta)$, where σ is still a general unifier.

$\underline{(a \lessdot_{?}{}_{?}\theta') \in Eq'_2}$**:** This case is analogous to the first case.

$\underline{(a \lessdot_{?}{}^{?}\theta') \in Eq'_2}$**:** It holds $\sigma(\, {}^{?}\theta'\,) \in \mathbf{grArg}(\, \sigma(\, a\,)\,)$. From this follows by Def. 1 that there is a type term $\theta' \in \mathbf{smArg}(\, {}^{?}\theta'\,)$ and $\sigma(\, a\,) = \sigma(\, \theta'\,)$.

This means for the pair $(a \lessdot_{?}{}^{?}\theta') \in Eq'_2$, where σ is a general unifier, there is a set of equations in Eq_{set} after the transformation with a pair $(a \doteq \theta')$, where σ is still a general unifier.

$\underline{(a \lessdot_{?} \theta') \in Eq'_2}$**:** It is obvious, that a general unifier of $(a \lessdot_{?} \theta')$ is also a general unifier of $(a \doteq \theta')$.

$\underline{(\theta \lessdot a) \in Eq'_2}$**,** $\underline{({}_{?}\theta \lessdot_{?} a) \in Eq'_2}$**:** The proof of these cases is analogous to the proof of case $(a \lessdot_{?}{}^{?}\theta') \in Eq'_2$

$\underline{({}^{?}\theta \lessdot_{?} a) \in Eq'_2}$**:** The proof of this case is analogous to the proof of case $(a \lessdot \theta') \in Eq'_2$.

$\underline{(\theta \lessdot_{?} a) \in Eq'_2}$**:** The proof of this case is analogous to the proof of case $(a \lessdot_{?}{}^{?}\theta') \in Eq'_2$

As in step four, the Cartesian product of the respective results is built, and there is one set of type term pairs of Eq_{set}, where σ is still a general unifier.

Step five: If σ is a general unifier of $\{\, a \doteq \theta\,\}$, then holds $\sigma(\, a[a \mapsto \theta]\,) = \sigma(\, \theta\,)$. From this follows that for all pairs $(\overline{\theta} \lessdot \overline{\theta}')\in Eq''$, $(\overline{\theta} \lessdot_{?} \overline{\theta}') \in Eq''$, and $(\overline{\theta} \doteq \overline{\theta}') \in Eq''$, respectively, with $\sigma(\overline{\theta}\,) \leq^* \sigma(\overline{\theta}'\,)$ $\sigma(\overline{\theta}'\,) \in \mathbf{grArg}(\sigma(\overline{\theta}\,))$, and $\sigma(\overline{\theta}\,) = \sigma(\overline{\theta}'\,)$, respectively, holds $\sigma(\overline{\theta}[a \mapsto \theta]\,) \leq^* \sigma(\overline{\theta}'[a \mapsto \theta]\,)$, $\sigma(\overline{\theta}'[a \mapsto \theta]\,) \in \mathbf{grArg}(\,\sigma(\overline{\theta}[a \mapsto \theta]\,)\,)$, and $\sigma(\overline{\theta}[a \mapsto \theta]\,) = \sigma(\overline{\theta}'[a \mapsto \theta]\,)$, respectively. This means that after step five σ is still a general unifier of the transformed set of type term pairs Eq'.

Step six: As no pair of type terms is transformed, the unifiers are maintained.

Step seven: In step seven the pairs of type terms which are in solved form are filtered. All other sets of pairs of type terms have no unifier, thus all unifiers are obtained.

We have proved that for an arbitrary given general unifier σ of

$$\{\, \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \,\}$$

σ is calculated by the algorithm. This means that the algorithm is complete.

As step four of the type unification algorithm is the only possibility, where the number of unifiers are multiplied, we can, according to Lemma 1 and Theorem 1, conclude as follows.

Corollary 1 (Finitary). *The type unification of* Java 5.0 *type terms with wildcards is finitary.*

In Example 6 is shown, how the problem of infinite chains is solved.

Corollary 2 (Termination). *The type unification algorithm terminates.*

Corollary 1 means that the open problem of [5], type unification in TEL, is also solved by our type unification algorithm. In TEL there are no wildcards. This means that in the type unification algorithm the differentiation between $\lessdot$, $\lessdot_?$, and $\doteq$ is unnecessary. Following this, the rules *reduceExt*, *reduceSup*, *reduceEq*, and *reduce2* and the functions **smArg** and **grArg** are also unnecessary. Furthermore in the finite closure there are only type terms of the form $C(\, a_1, \ldots, a_n \,)$ where the a_i are type variables, which means that the adapt rules are also unnecessary. Finally in TEL in the ordering $<$ for $\theta < \theta'$ holds $\mathsf{TVar}(\,\theta\,) \subseteq \mathsf{TVar}(\,\theta'\,)$, while is Java 5.0 holds $\mathsf{TVar}(\,\theta'\,) \subseteq \mathsf{TVar}(\,\theta\,)$. This means that infinite chains in TEL has lower bounds, while infinite chains in Java 5.0 has upper bounds. Let us consider again the example from section 3.1. In TEL for `List(a)` $\leq$ `myLi(a,b)` holds:

$$\texttt{List(a)} \leq \texttt{myLi(a,List(a))} \leq \texttt{myLi(a,myLi(a,List(a)))} \leq \ldots$$

In contrast in Java 5.0 for `myLi<b, a>` $<$ `List<a>` holds:

$$\ldots \leq^* \texttt{myLi<}_?\texttt{myLi<}_?\texttt{List<a>, a>, a>} \leq^* \texttt{myLi<}_?\texttt{List<a>, a>} \leq^* \texttt{List<a>}$$

We close with a type unfication example in TEL. The corresponding example in Java 5.0 is given in Example 6.

Example 7. Let $\texttt{List}(\,\texttt{a}\,) < \texttt{myLi}(\,\texttt{b}, \texttt{a}\,)$ and the start configuration

$$\{\, \texttt{List}(\,\texttt{List}(\,\texttt{Integer}\,)\,) \lessdot \texttt{List}(\,\texttt{x}\,)\,\}$$

be given. In the first step the reduce 1 rule is applied: $\{\, \texttt{List}(\,\texttt{Integer}\,) \lessdot \texttt{x}\,\}$. With the fourth step we get the result:

$$\{\, \{\, \texttt{x} \mapsto \texttt{List}(\,\texttt{Integer}\,)\,\}, \{\, \texttt{x} \mapsto \texttt{myLi}(\,\texttt{b}, \texttt{Integer}\,)\,\}\,\}.$$

All other infinite numbers of unifiers are instances of these general unifiers (e.g. $\{\, \texttt{x} \mapsto \texttt{myLi}(\,\texttt{List}(\,\texttt{Integer}\,), \texttt{Integer}\,)\,\}$ or $\{\, \texttt{x} \mapsto \texttt{myLi}(\,\texttt{myList}(\,\texttt{b}, \texttt{Integer}\,), \texttt{Integer}\,)\,\}$).

4 Conclusion and Outlook

In this paper we presented a unification algorithm, which solves the type unification problem of Java 5.0 type terms with wildcards. Although the Java 5.0 subtyping ordering contains infinite chains, we showed that the type unification is finitary. This means that we solved the open problem from [5].

The Java 5.0 type unification is the base of the Java 5.0 type inference [3], as the usual unification is the base of type inference in functional programming languages.

With the type unification algorithm, which we have presented in this paper, it would be possible to complete the type inference algorithm of TEL [5] respectively to extend the type system of PROTOS-L [8] by allowing explicit subtype relationships between polymorphic type constructors.

References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification, 3rd edn. The Java series. Addison-Wesley, Reading (2005)
2. Plümicke, M., Bäuerle, J.: Typeless Programming in Java 5.0. In: Gitzel, R., Aleksey, M., Schader, M., Krintz, C. (eds.) 4th International Conference on Principles and Practices of Programming in Java. ACM International Conference Proceeding Series, pp. 175–181. Mannheim University Press (August 2006)
3. Plümicke, M.: Typeless Programming in Java 5.0 with wildcards. In: Amaral, V., Veiga, L., Marcelino, L., Cunningham, H.C. (eds.) 5th International Conference on Principles and Practices of Programming in Java. ACM International Conference Proceeding Series, pp. 73–82. ACM Press, New York (September 2007)
4. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proc. 9th Symposium on Principles of Programming Languages (1982)
5. Smolka, G.: Logic Programming over Polymorphically Order-Sorted Types. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany (May 1989)
6. Hanus, M.: Parametric order-sorted types in logic programming. In: Abramsky, S. (ed.) TAPSOFT 1991, CCPSD 1991, and ADC-Talks 1991. LNCS, vol. 494, pp. 181–200. Springer, Heidelberg (1991)
7. Hill, P.M., Topor, R.W.: A Semantics for Typed Logic Programs. In: Pfenning, F. (ed.) Types in Logic Programming, pp. 1–62. MIT Press, Cambridge (1992)
8. Beierle, C.: Type inferencing for polymorphic order-sorted logic programs. In: International Conference on Logic Programming, pp. 765–779 (1995)
9. Plümicke, M.: OBJ–P The Polymorphic Extension of OBJ–3. PhD thesis, University of Tuebingen, WSI-99-4 (1999)
10. Plümicke, M.: Type unification in Generic–Java. In: Kohlhase, M. (ed.) Proceedings of 18th International Workshop on Unification (UNIF 2004) (July 2004)
11. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems 4, 258–282 (1982)

Testing Relativised Uniform Equivalence under Answer-Set Projection in the System $\texttt{cc}\top$*

Johannes Oetsch[1], Martina Seidl[2], Hans Tompits[1], and Stefan Woltran[1]

[1] Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,tompits}@kr.tuwien.ac.at,
woltran@dbai.tuwien.ac.at

[2] Institut für Softwaretechnik, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
seidl@big.tuwien.ac.at

Abstract. The system $\texttt{cc}\top$ is a tool for testing correspondence between propositional logic programs under the answer-set semantics with respect to different refined notions of program correspondence. The underlying methodology of $\texttt{cc}\top$ is to reduce a given correspondence problem to the satisfiability problem of quantified propositional logic and to employ extant solvers for the latter language as back-end inference engines. In a previous version of $\texttt{cc}\top$, the system was designed to test correspondence between programs based on *relativised strong equivalence under answer-set projection*. Such a setting generalises the standard notion of strong equivalence by taking the alphabet of the context programs as well as the projection of the compared answer sets to a set of designated output atoms into account. This paper outlines a newly added component of $\texttt{cc}\top$ for testing similarly parameterised correspondence problems based on *uniform equivalence*.

1 Motivation and General Information

An important issue in software development is to determine whether two encodings of a given problem are equivalent, i.e., whether they yield the same result on a given problem instance. Depending on the context of problem representations, different definitions of "equivalence" are useful and desirable. The system $\texttt{cc}\top$ [1] (short for "correspondence-checking tool") is devised as a checker for a broad range of different such comparison relations defined between *disjunctive logic programs* (DLPs) *under the answer-set semantics* [2]. In a previous version of $\texttt{cc}\top$, the system was designed to test correspondence between logic programs based on *relativised strong equivalence under answer-set projection*. Such a setting generalises the standard notion of strong equivalence [3] by taking the alphabet of the context programs as well as the projection of the compared answer sets to a set of designated output atoms into account [4]. The latter feature

* This work was partially supported by the Austrian Science Fund (FWF) under grant P18019. The second author was also supported by the Austrian Federal Ministry of Transport, Innovation, and Technology (BMVIT) and the Austrian Research Promotion Agency (FFG) under grant FIT-IT-810806.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 241–246, 2009.

reflects the common use of local (hidden) variables, which may be employed in submodules, but which are ignored in the final computation.

In this paper, we outline a newly added component of $\texttt{cc}\top$ for testing similarly parameterised correspondence problems but generalising *uniform equivalence* [5]—that is, we deal with a component of $\texttt{cc}\top$ for testing *relativised uniform equivalence under answer-set projection*. This recent notion [6] is less restrained than its strong counterpart, along with a slightly lower complexity (provided that the polynomial hierarchy does not collapse). However, in general, it is still outside the feasible computational means of propositional answer-set solvers (again under the proviso that the polynomial hierarchy does not collapse). Yet, like relativised strong equivalence with projection, it can be efficiently reduced to the satisfiability problem of quantified propositional logic, an extension of classical propositional logic characterised by the condition that its sentences, generally referred to as *quantified Boolean formulas* (QBFs), are permitted to contain quantifications over atomic formulas. The architecture of $\texttt{cc}\top$ takes advantage of this and uses extant solvers for quantified propositional logic as back-end reasoning engines.

2 Background

Propositional disjunctive logic programs (DLPs) are finite sets of rules of the form

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \mathrm{not}\, a_{m+1}, \ldots, \mathrm{not}\, a_n, \tag{1}$$

$n \geq m \geq l \geq 0$, where all a_i are propositional atoms from some fixed universe $\mathcal{U}$ and not denotes default negation. A *fact* is a rule of form (1) with $l = m = n = 1$. For facts, we usually omit the symbol $\leftarrow$.

An *interpretation* I is a *model* of a program P, denoted by $I \models P$, iff for every rule from P (as defined above), it holds that, whenever $\{a_{l+1}, \ldots, a_m\} \subseteq I$ and $\{a_{m+1}, \ldots, a_n\} \cap I = \emptyset$, then $\{a_1, \ldots, a_l\} \cap I \neq \emptyset$.

Following Gelfond and Lifschitz [2], an interpretation I is an *answer set* of a program P iff it is a minimal model of the *reduct* P^I, resulting from P by (i) deleting all rules containing default negated atoms $\mathrm{not}\, a$ such that $a \in I$, and (ii) deleting all default negated atoms in the remaining rules. The collection of all answer sets of a program P is denoted by $\mathcal{AS}(P)$.

In order to semantically compare programs, different notions of equivalence have been introduced in the context of the answer-set semantics. Two programs, P and Q, are *strongly equivalent* iff, for any program R, $AS(P \cup R) = AS(Q \cup R)$; they are *uniformly equivalent* iff, for any set F of facts, $AS(P \cup F) = AS(Q \cup F)$. While strong equivalence is relevant for program optimisation and modular programming in general [7,8,9], uniform equivalence is useful in the context of hierarchically structured program components, where lower-layered components provide input for higher-layered ones. In abstracting from strong and uniform equivalence, Eiter *et al.* [4] introduced the notion of a *correspondence problem*, which allows to specify (i) a *context*, i.e., a class of programs that can be added to the programs under consideration, and (ii) the comparison relation that has to hold between the answer sets of the extended programs. Following Eiter *et al.* [4], we focus here on correspondence problems where the comparison relation enhances the standard subset or set-equality relation by incorporating projection to a given set of atoms. The context, on the other hand, contains all programs that are sets of facts over some set A of atoms, identified with the power set 2^A of A.

Thus, the concrete formal realisation of relativised uniform equivalence with projection is as follows [6]: Consider a quadruple $\Pi = (P, Q, 2^A, \odot_B)$, where P, Q are programs, A, B are sets of atoms, $\odot \in \{\subseteq, =\}$, and $\mathcal{S} \ \odot_B \ \mathcal{S}'$ stands for $\{I \cap B \mid I \in \mathcal{S}\} \odot \{J \cap B \mid J \in \mathcal{S}'\}$. Π is called a *propositional query equivalence problem* (PQEP) if $\odot_B$ is given by $=_B$, and a *propositional query inclusion problem* (PQIP) if $\odot_B$ is given by $\subseteq_B$. We say that Π *holds* iff, for each $F \in 2^A$, $AS(P \cup F) \odot_B AS(Q \cup F)$. Note that $(P, Q, 2^A, =_B)$ holds iff $(P, Q, 2^A, \subseteq_B)$ and $(Q, P, 2^A, \subseteq_B)$ jointly hold. We also refer to A as the *context set* and to B as the *projection set*.

For illustration, consider the programs

$$P = \{\mathit{sad} \vee \mathit{happy} \leftarrow;\ \mathit{sappy} \leftarrow \mathit{sad}, \mathit{happy};\ \mathit{confused} \leftarrow \mathit{sappy}\} \quad \text{and}$$
$$Q = \{\mathit{sad} \leftarrow \text{not } \mathit{happy};\ \mathit{happy} \leftarrow \text{not } \mathit{sad};\ \mathit{confused} \leftarrow \mathit{sad}, \mathit{happy}\},$$

which express some knowledge about the "moods" of a person, where P uses an auxiliary atom sappy. The programs can be seen as queries over a propositional database consisting of facts from, e.g., $\{\mathit{happy}, \mathit{sad}\}$. For the output, it would be natural to consider the common intensional atom $\mathit{confused}$. We thus consider $\Pi = (P, Q,\, 2^A, =_B)$ as a suitable PQEP, specifying $A = \{\mathit{happy}, \mathit{sad}\}$ and $B = \{\mathit{confused}\}$. It is a straightforward matter to check that Π, defined in this way, holds.

3 System Specifics

As pointed out in Section 1, the overall approach of cc⊤ is to reduce PQEPs and PQIPs to the satisfiability problem of quantified propositional logic and to use extant solvers [10] for the latter language as back-end inference engines for evaluating the resulting formulas. The reductions required for this approach are described by Oetsch *et al.* [6] but cc⊤ employs additional optimisations [11]. The overall application framework for cc⊤ is depicted in Fig. 1. The system takes as input two programs, P and Q, and two sets of atoms, A and B. Command-line options select between two kinds of reductions, a direct one or an optimised one, and whether the programs are compared as a PQIP or a PQEP. Detailed invocation syntax can be requested with option '`-h`'.

Next, let us turn our attention to the concrete usage of cc⊤. The syntax of the programs is the basic DLV syntax.[1] In this syntax, the two programs P and Q from the above example look as follows:

$$P = \left\{\begin{array}{l}\texttt{sad v happy.}\\ \texttt{sappy :- sad, happy.}\\ \texttt{confused :- sappy.}\end{array}\right. \qquad Q = \left\{\begin{array}{l}\texttt{sad :- not happy.}\\ \texttt{happy :- not sad.}\\ \texttt{confused :- sad, happy.}\end{array}\right.$$

Let us assume that the two programs are stored in the files `P.dl` and `Q.dl`. The two sets A and B from the example are written as comma separated lists within brackets:

[1] See `http://www.dlvsystem.com/` for details about DLV.

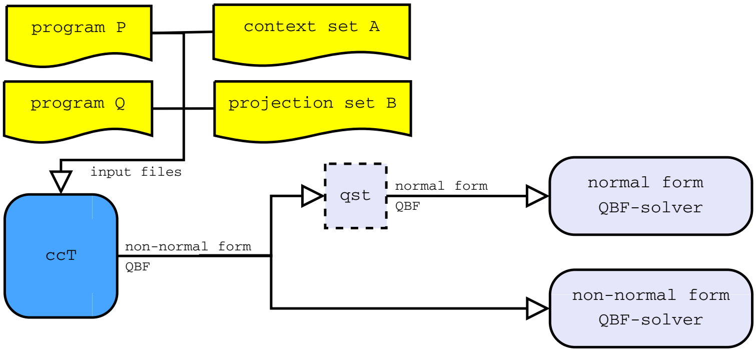


Fig. 1. Overall application framework for `cc⊤`

context set A: `(happy, sad)`,
projection set B: `(confused)`.

We assume them to be stored in files A and B. The concrete invocation syntax for translating the problem $\Pi = (P, Q, 2^A, =_B)$ into a corresponding QBF is

```
ccT -u -e P.dl Q.dl A B
```

where the command-line options '`-u`' and '`-e`' evince that we want to check for a notion of *u*niform *e*quivalence. To check for uniform inclusion, replace '`-e`' by '`-i`' or omit the parameter.

The output will be written directly to the standard-output device from where it can serve as input for QBF-solvers. Since `cc⊤` does not output QBFs in a specific normal form, for using solvers requiring normal-form QBFs, the additional normaliser `qst` [12] is employed. Finally, `cc⊤` is developed entirely in ANSI C; hence, it is highly portable. The parser for the input data was written using LEX and YACC. Further information about `cc⊤` is available at

`http://www.kr.tuwien.ac.at/research/systems/ccT/`.

Experimental evaluations using different QBF-solvers are reported in a companion paper [11].

4 cc⊤ on Stage

In this section, we give a brief and, for space reasons, rather informal discussion on the application of `cc⊤` for verification and debugging needs in the context of a logic programming course at our university. This is not only to make the concept of correspondence checking within a refined framework more tangible, but also to show a concrete application field. As a subtask in this course, the students had to model an air-conditioning system consisting of components for cooling and heating, as well as a valve and a switch element. More specifically, they were given a detailed description of the desired input/output behaviour of the components and the system as a whole, and they had to develop logic programs that comply with that specification. Without going into details, such a specification could state that the input of, e.g., a heating component consists of an airstream, which can be 0 (air does not float) or 1 (air floats), and has an associated temperature (an integer from a certain range) as well as a control parameter (also an integer) to control the heating power. Analogously to the input airstream, a heater has an output airstream. Now, the specification determines the behaviour of the component with respect to the output airstream conditioned by the input airstream and the control parameter.

A straightforward strategy to verify a student's solution is the following: (i) write a sample solution that correctly implements the specification, (ii) define test cases, i.e., sets of facts representing the input for a component, and (iii) compare the output of our sample solution against the output of the student's component. This method, used in previous years and implemented by a more or less simple script, is obviously sound but not complete with respect to detecting potential flaws. Here comes ccT into the play: this verification problem can be stated as a PQEP[2], where the context set consist of the atoms that constitute the input and the projection set contains the atoms that represent the output of a component (thus allowing the students an unrestricted use of additional atoms in their programs). Hence, we have a sound and complete method for verification at hand. We employed this approach last winter semester for evaluating the submitted exercises, and it compared favourably to the old method.

Two things were necessary to obtain reasonable run-times for evaluating the QBFs, however: First, we had to restrict the context class, and second, we added additional constraints to the programs to impose some restrictions on the input of the components, like specifying not more than one input value for an airstream temperature. The latter point is also to make the test more fair. Albeit we loose completeness in the sense from above this way, we are able to verify thousands of test cases implicitly with the ccT-approach compared to only 10 to 20 test cases with the old script. Also, a direct comparison of the results between the two test approaches is very encouraging: all errors detected by the script were also detected by the ccT-approach, while 26 (out of 200) components were classified as correct by the script but as non-equivalent to our sample solution by ccT. It is worth mentioning that on the considered problem instances the solver qpro [13] showed, with a median run-time of 2.54 seconds, excellent performance.

5 Conclusion

In this paper, we presented the architecture and system specifics of a new component of the tool ccT for testing parameterised correspondence problems based on *uniform equivalence* for disjunctive logic programs under the answer-set semantics. The correspondence problems are efficiently compiled to quantified Boolean formulas for which many solvers have been implemented. As related work, we mention the system DLPEQ [14] for deciding ordinary equivalence, which is based on a reduction to logic programs, and

[2] The programs under consideration are not propositional, i.e., they contain variables. Nevertheless, the domain, i.e., the set of constants that can occur in the programs, is finite, and such programs can always be treated as shorthands for the respective propositional programs.

the system SELP [15] for checking strong equivalence, which is based on a reduction to classical logic quite in the spirit of our implementation approach. We successfully applied cc$\top$ for the verification of students' programs obtained from a laboratory course on logic programming at our university. Future work includes extending our methods to non-ground logic programs, which are important in practical applications.

References

1. Oetsch, J., Seidl, M., Tompits, H., Woltran, S.: ccT: A Tool for Checking Advanced Correspondence Problems in Answer-Set Programming. In: Proc. of the 15th International Conference on Computing (CIC 2006), pp. 3–10. IEEE Computer Society Press, Los Alamitos (2006)
2. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
3. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM TOCL 2(4), 526–541 (2001)
4. Eiter, T., Tompits, H., Woltran, S.: On Solution Correspondences in Answer Set Programming. In: Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 97–102 (2005)
5. Eiter, T., Fink, M.: Uniform equivalence of logic programs under the stable model semantics. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 224–238. Springer, Heidelberg (2003)
6. Oetsch, J., Tompits, H., Woltran, S.: Facts do not Cease to Exist Because They are Ignored: Relativised Uniform Equivalence with Answer-Set Projection. In: Proc. of the 22nd National Conference on Artificial Intelligence (AAAI 2007), pp. 458–464. AAAI Press, Menlo Park (2007)
7. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS, vol. 2923, pp. 87–99. Springer, Heidelberg (2003)
8. Pearce, D.: Simplifying Logic Programs under Answer Set Semantics. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 210–224. Springer, Heidelberg (2004)
9. Lin, F., Chen, Y.: Discovering Classes of Strongly Equivalent Logic Programs. In: Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 516–521 (2005)
10. Le Berre, D., Narizzano, M., Simon, L., Tacchella, A.: The second QBF solvers comparative evaluation. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 376–392. Springer, Heidelberg (2005)
11. Oetsch, J., Seidl, M., Tompits, H., Woltran, S.: An Extension of the System cc$\top$ for Testing Relativised Uniform Equivalence under Answer-Set Projection. In: Proc. of the 16th International Conference on Computing (CIC 2007) (to appear, 2007)
12. Zolda, M.: Comparing Different Prenexing Strategies for Quantified Boolean Formulas, Master's Thesis, Vienna University of Technology (2004)
13. Egly, U., Seidl, M., Woltran, S.: A Solver for QBFs in Nonprenex Form. In: Proc. of the 17th European Conference on Artificial Intelligence (ECAI 2006), pp. 477–481 (2006)
14. Oikarinen, E., Janhunen, T.: Verifying the Equivalence of Logic Programs in the Disjunctive Case. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS, vol. 2923, pp. 180–193. Springer, Heidelberg (2003)
15. Chen, Y., Lin, F., Li, L.: SELP - A system for studying strong equivalence between logic programs. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS, vol. 3662, pp. 442–446. Springer, Heidelberg (2005)

spock: A Debugging Support Tool for Logic Programs under the Answer-Set Semantics*

Martin Gebser[1], Jörg Pührer[2], Torsten Schaub[1],
Hans Tompits[2], and Stefan Woltran[2]

[1] Institut für Informatik, Universität Potsdam,
August-Bebel-Straße 89, D-14482 Potsdam, Germany
{gebser,torsten}@cs.uni-potsdam.de

[2] Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9–11, A–1040 Vienna, Austria
{puehrer,tompits}@kr.tuwien.ac.at,
woltran@dbai.tuwien.ac.at

Abstract. Answer-set programming (ASP) is an emerging logic-programming paradigm that strictly separates the description of a problem from its solving methods. Despite its semantic elegance, ASP suffers from a lack of support for program developers. In particular, tools are needed that help engineers in detecting erroneous parts of their programs. Unlike in other areas of logic programming, applying tracing techniques for debugging logic programs under the answer-set semantics seems rather unnatural, since employing imperative solving algorithms would undermine the declarative flavour of ASP. In this paper, we present the system spock, a debugging support tool for answer-set programs making use of ASP itself. The implemented techniques maintain the declarative nature of ASP within the debugging process and are independent of the actual computation of answer sets.

1 General Information

Answer-set programming (ASP) [1] has become an important logic-programming paradigm for declarative problem solving, incorporating fundamental concepts of nonmonotonic reasoning. A major reason why ASP has not yet found a more widespread popularity as a problem-solving technique, however, is its lack of suitable *engineering tools* for developing programs. In particular, realising tools for *debugging* answer-set programs is a clearly recognised issue in the ASP community, and several approaches in this direction have been proposed in recent years [2,3,4,5].

From a theoretical point of view, the nonmonotonicity of answer-set programs is an aggravating factor for detecting sources of errors, since every rule of a program might significantly influence the resulting answer sets. On the other hand, applying tracing techniques for debugging logic programs under the answer-set semantics seems rather unnatural, since employing imperative solving algorithms would undermine the declarative flavour of ASP.

* This work was partially supported by the Austrian Science Fund (FWF) under project P18019.

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 247–252, 2009.

In this paper, we discuss the main features of the system `spock` [6], which supports developers of answer-set programs in locating errors in their programs by exploiting the declarative nature of ASP itself, but being independent of specific ASP solvers. The name "`spock`" makes reference to the fact that detecting errors is done by means of logic, just like the popular Vulcan of Star Trek fame.

The theoretical background of the implemented methods was introduced in previous work [5], exploiting and extending a *tagging technique* as used by Delgrande et al. [7] for compiling ordered logic programs into standard ones. In our approach, a program to debug, Π, is augmented with dedicated meta-atoms, called *tags*, serving two purposes: Firstly, they allow for controlling and manipulating the applicability of rules, and secondly, tags occurring in the answer sets of the extended program reflect various properties of Π. Our tool implements the tagging process and further related translations for a program Π to debug, allowing for an extrapolation of non-existing answer sets in combination with explanations why an interpretation is not an answer set of Π.

For illustration of the debugging questions addressed, consider the problem of inviting guests to a party when it is known that some of them would appear only if certain others do or do not attend the festivity. An instance of such a setting is encoded in program Π_{inv}, where each atom represents the appearing of a potential party visitor:

$$\begin{array}{rclcrcl}
r_1 & = & jim \leftarrow uhura, & \quad & r_4 & = & chekov \leftarrow not\ bones, \\
r_2 & = & jim \leftarrow not\ chekov, & & r_5 & = & bones \leftarrow jim, \\
r_3 & = & uhura \leftarrow chekov, not\ scotty, & & r_6 & = & scotty \leftarrow not\ uhura.
\end{array}$$

This program has two answer sets, viz., $\{chekov, scotty\}$ and $\{bones, jim, scotty\}$. Assume that Sulu, the programmer, is quite perplexed by this result, wondering why there is a scenario where only Chekov and Scotty, who merely have a neutral relation to each other rather than a friendship, attend. On the other hand, he is astonished as there is no possibility such that Uhura and Jim can jointly be invited. With the help of the tool `spock`, reasons for such mismatches between the expected and the actual semantics of a program can be found.

2 Background

2.1 Answer-Set Programs

A (*normal*) *logic program* (over an alphabet $\mathcal{A}$) is a finite set of rules of the form

$$a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n, \tag{1}$$

where a and $b_i, c_j \in \mathcal{A}$ are atoms, for $0 \leq i \leq m$, $0 \leq j \leq n$. A *literal* is an atom a or its negation $not\ a$. For a rule r as in (1), let $head(r) = a$ be the *head* of r and $body(r) = \{b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n\}$ the *body* of r. Furthermore, we define $body^+(r) = \{b_1, \ldots, b_m\}$ and $body^-(r) = \{c_1, \ldots, c_n\}$. For a logic program Π, a set X of atoms is an *answer set* of Π iff X is a minimal model of $\{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}$. For uniformity, we assume that any integrity constraint $\leftarrow body(r)$ is expressed as a rule $w \leftarrow body(r), not\ w$, where w is a globally new atom. Moreover, we allow nested expressions of form $not\ not\ a$, where a is some atom, in the body of rules. Such rules are identified with normal rules in which $not\ not\ a$ is replaced by $not\ a^\star$, where $a^\star$ is a globally new atom, together with an additional rule $a^\star \leftarrow not\ a$.

2.2 Tagging-Based Debugging

In what follows, we sketch the theoretical principles underlying our system spock. For a more detailed discussion, we refer to Brain et al. [5]. The main idea of tagging is to split the head from the body, for each rule in a program, and thereby to intervene into the applicability of rules. After this division, tags are installed for triggering rules. This way, the formation of answer sets can be controlled, and tags in the answer sets of the transformed (or tagged) program reflect inherent properties of the original program.

Technically, a program Π (over alphabet $\mathcal{A}$) to debug is rewritten into a program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over an extended alphabet $\mathcal{A}^+$. Let Π be a logic program over $\mathcal{A}$ and consider a bijection n, assigning to each rule r over $\mathcal{A}$ a unique name n_r. Then, the program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over $\mathcal{A}^+$ consists of the following rules, for $r \in \Pi$, $b \in body^+(r)$, and $c \in body^-(r)$:

$$head(r) \leftarrow \mathsf{ap}(n_r), not\ \mathsf{ko}(n_r), \tag{2}$$

$$\mathsf{ap}(n_r) \leftarrow \mathsf{ok}(n_r), body(r), \tag{3}$$

$$\mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), not\ b, \tag{4}$$

$$\mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), not\ not\ c, \tag{5}$$

$$\mathsf{ok}(n_r) \leftarrow not\ \overline{\mathsf{ok}}(n_r). \tag{6}$$

The tags $\mathsf{ap}(n_r)$ and $\mathsf{bl}(n_r)$ express whether a rule r is applicable or blocked, respectively, while the *control tags* $\mathsf{ko}(n_r)$, $\mathsf{ok}(n_r)$, and $\overline{\mathsf{ok}}(n_r)$ are used for manipulating the application of r. Intuitively, the rules of Π are split into rules of forms (2) and (3), separating the applicability of a rule from the actual occurrence of the respective rule head in an interpretation. Analogously, rules of forms (4) and (5) elicit which rules are blocked. Tags stating whether rule r is applicable or blocked are only derived if $\mathsf{ok}(n_r)$ holds, which is by default the case, as expressed by rules of form (6).

We call $\mathcal{T}_{\mathsf{K}}[\Pi]$ the *kernel tagging* of Π, since it serves as a basic submodule for more enhanced programs facilitating certain debugging requests. One such extension scenario is the extrapolation of non-existing answer sets of a program Π over $\mathcal{A}$. Using further translations, $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$ [5], the occurrence of *abnormality tags*, $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$, respectively, in an answer set X^+ of the transformed program provides information why an interpretation $X = X^+ \cap \mathcal{A}$ is not an answer set of Π. Here, we make use of the Lin-Zhao theorem [8], which qualifies answer sets as models of the *completion* [9] and the *loop formulas* of a program. In particular, the program-oriented abnormality tag $\mathsf{ab}_p(n_r)$ indicates that rule r is applicable but not satisfied with respect to an interpretation. The completion-oriented abnormality tag $\mathsf{ab}_c(a)$ signals that a is in the considered interpretation but all rules having a as head are blocked. Finally, the presence of a loop-oriented abnormality tag $\mathsf{ab}_l(a)$ indicates that the derivation of atom a might recursively depend on a itself and, therefore, violate the minimality criterion for answer sets. Note that all transformations used are polynomial in the size of the input program and can be constructed for all programs under consideration, even for programs without answer sets.

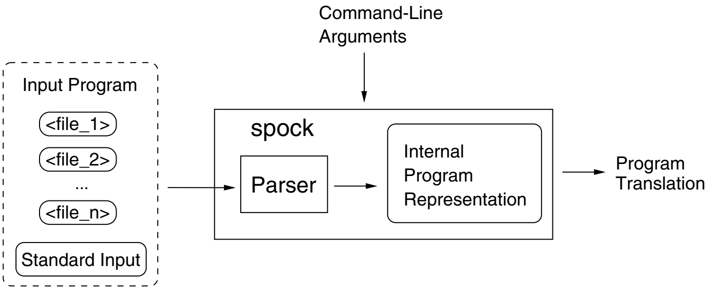


Fig. 1. Data flow of program translations

3 System

spock is a command-line oriented tool, written in Java 5.0 and published under the GNU general public license [10]. It is publicly available at

`http://www.kr.tuwien.ac.at/research/debug`

as a jar-package including binaries and sources.

The data flow for all transformations is depicted by Fig. 1. First, the input program is parsed and represented in an internal data structure. Then, the actual program transformation is performed, as specified by command-line arguments.

The tagging technique uses labels to refer to individual rules. Therefore, we allow the programmer to add labels to the rules of the program to debug. As this requires an extension of the program syntax, spock offers an interface to dlv [11] and lparse/smodels [12] for computing answer sets of labelled programs.

For illustration of the debugging process, reconsider program Π_{inv}, having the answer sets $X_1 = \{chekov, scotty\}$ and $X_2 = \{bones, jim, scotty\}$, and assume that it is stored in file FILE. The kernel tagging $\mathcal{T}_{\mathsf{K}}[\Pi_{inv}]$ is then obtained by the call

```
java -jar spock.jar -k FILE .
```

By piping the result of the command to an answer-set solver, we obtain the answer sets

$$X_1^+ = X_1 \cup \{\mathsf{ap}(n_{r_4}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_5})\} \cup \mathsf{OK} \text{ and}$$
$$X_2^+ = X_2 \cup \{\mathsf{ap}(n_{r_2}), \mathsf{ap}(n_{r_5}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4})\} \cup \mathsf{OK},$$

where $\mathsf{OK} = \{\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3}), \mathsf{ok}(n_{r_4}), \mathsf{ok}(n_{r_5}), \mathsf{ok}(n_{r_6})\}$, extending X_1 and X_2 by information about the applicability of rules. E.g., the presence of $\mathsf{ap}(n_{r_4})$ in X_1^+ indicates that rule r_4 is applicable with respect to X_1, and hence $chekov \in X_1$ but $bones \notin X_1$, while $\mathsf{bl}(n_{r_3}) \in X_1^+$ indicates that r_3 is blocked with respect to X_1. This is because $scotty \in X_1$.

The flags '`-expo`', '`-exco`', and '`-exlo`' activate the extrapolation translations $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$, respectively. Instead of using all three flags simultaneously, setting '`-ex`' produces the union of the resulting programs. Furthermore, in order to restrict the scope of transformation $\mathcal{T}_{\mathsf{P}}$ to a subprogram Π' (respectively, translations $\mathcal{T}_{\mathsf{C}}, \mathcal{T}_{\mathsf{L}}$ to sets A_{C}, A_{L} of atoms), the names of the considered rules (respectively, atoms) can be explicitly stated in a comma-separated list following the '`-exrules=`' (resp., '`-exatomsC=`' and '`-exatomsL=`') flag. Finally, spock allows for computing only abnormality-minimum answer sets by means of dlv-specific weak constraints. The flags '`-minab`', '`-minabp`', '`-minabc`', or '`-minabl`' make spock output weak constraints for minimising all abnormality tags, program-oriented abnormality tags, completion-oriented abnormality tags, or loop-oriented abnormality tags, respectively.

As for our example, recall that Sulu wanted to know why there is no chance for Uhura and Jim to attend the same party. Therefore, we add the constraints $\leftarrow$ *not uhura* and $\leftarrow$ *not jim* to Π_{inv}. Let file FILE2 contain the overall program, which does not have answer sets. The (optimal) answer sets of the tagged program obtained by the call

```
java -jar spock.jar -k -ex -exrules=r1,r2,r3,r4,r5,r6
                     -minab FILE2,
```

projected to the atoms occurring in Π_{inv} and the abnormality tags, are given by $\{\mathsf{ab}_c(chekov), bones, chekov, jim, uhura\}$, $\{\mathsf{ab}_c(uhura), bones, jim, uhura\}$, and $\{\mathsf{ab}_p(n_{r_5}), chekov, jim, uhura\}$, indicating that $\{bones, chekov, jim, uhura\}$ is not an answer set of Π_{inv} because atom *chekov* is not supported. Likewise, *uhura* is not supported with respect to $\{bones, jim, uhura\}$. Finally, $\{chekov, jim, uhura\}$ is not an answer set as it does not satisfy rule r_5.

4 Discussion and Related Work

In this paper, we presented spock, a prototype implementation of a debugging support tool for answer-set programs. The implemented methodology relies on theoretical results of previous work [5] and is based on the idea that programs to be debugged are translated into other programs having answer sets that offer debugging-relevant information about the original programs. After an initial kernel transformation, we get insight into the applicability of rules with respect to individual answer sets. In a further step, the system allows for identifying causes why interpretations are not answer sets. Here, spock distinguishes between abnormalities due to missing or spare atoms, or atoms whose presence in an interpretation is self-caused. In order to restrict the amount of information returned to the programmer, standard ASP optimisation techniques can be used to focus on interpretations with a minimal number of abnormalities. In addition to the tagging technique described here, spock also supports another approach towards debugging answer-set programs based on meta-programming [13,14]. Future work includes the integration of further aspects of the translation approach and the design of a graphical user interface to ease the use of the features spock provides.

Implementations of related techniques include smdebug [3], a prototype debugger focusing on odd-cycle-free inconsistent programs. The system is designed to find minimal sets of constraints, restoring consistency when removed from a program. Brain and

De Vos [2] present the system *IDEAS*, implementing two query algorithms addressing the questions why a set of literals is true with respect to some or false with respect to all answer sets of a program. Both algorithms are procedural and similar to the ones used in ASP solvers, suggesting that an approach using program-level transformations would be more practical. Pontelli and Son [4] developed a preliminary implementation for their adoption of so-called *justifications* [15,16] to the problem of debugging answer-set programs. Their system returns visual output in form of graphs explaining why atoms are (not) present in an answer set.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Brain, M., De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In: Proc. ASP 2005, EUR Workshop Proceedings (CEUR-WS.org), vol. 142, pp. 141–152 (2005)
3. Syrjänen, T.: Debugging Inconsistent Answer Set Programs. In: Proc. NMR 2006, pp. 77–83 (2006)
4. Pontelli, E., Son, T.C.: Justifications for Logic Programs under Answer Set Semantics. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 196–210. Springer, Heidelberg (2006)
5. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
6. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: That is Illogical Captain! – The Debugging Support Tool spock for Answer-Set Programs: System Description. In: Proc. SEA 2007, pp. 71–85 (2007)
7. Delgrande, J., Schaub, T., Tompits, H.: A Framework for Compiling Preferences in Logic Programs. Theory and Practice of Logic Programming 3, 129–187 (2003)
8. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. Artificial Intelligence 157, 115–137 (2004)
9. Clark, K.: Negation as Failure. In: Logic and Data Bases, pp. 293–322. Plenum Press (1978)
10. GNU General Public License – Version 2, June 1991. Free Software Foundation Inc. (1991), `http://www.gnu.org/copyleft/gpl.html`
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7, 499–562 (2006)
12. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence 138, 181–234 (2002)
13. Pührer, J.: On Debugging of Propositional Answer-Set Programs. Master's thesis, Vienna University of Technology, Austria (2007)
14. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A Meta-Programming Technique for Debugging Answer-Set Programs. In: Proc. AAAI 2008 (2008)
15. Roychoudhury, A., Ramakrishnan, C., Ramakrishnan, I.: Justifying Proofs using Memo Tables. In: Proc. PPDP 2000, 178–189 (2000)
16. Specht, G.: Generating Explanation Trees even for Negations in Deductive Database Systems. In: Proc. LPE 1993, pp. 8–13 (1993)

Author Index