

Herbert Kuchen (Ed.)

LNCS 6816

# Functional and Constraint Logic Programming

20th International Workshop, WFLP 2011  
Odense, Denmark, July 2011  
Proceedings



Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Alfred Kobsa

*University of California, Irvine, CA, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*TU Dortmund University, Germany*

Madhu Sudan

*Microsoft Research, Cambridge, MA, USA*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max Planck Institute for Informatics, Saarbruecken, Germany*

Herbert Kuchen (Ed.)

# Functional and Constraint Logic Programming

20th International Workshop, WFLP 2011  
Odense, Denmark, July 19, 2011  
Proceedings

Volume Editor

Herbert Kuchen  
Westfälische Wilhelms-Universität Münster  
Institut für Wirtschaftsinformatik  
Leonardo-Campus 3, 48149 Münster, Germany  
E-mail: kuchen@uni-muenster.de

ISSN 0302-9743 e-ISSN 1611-3349  
ISBN 978-3-642-22530-7 e-ISBN 978-3-642-22531-4  
DOI 10.1007/978-3-642-22531-4  
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011931683

CR Subject Classification (1998): F.4, F.3.2, D.3, I.2.2-5, I.1

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

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

*Typesetting:* Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media ([www.springer.com](http://www.springer.com))

# Preface

This volume contains the proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), held in Odense, Denmark, July 19, 2011 at the University of Southern Denmark. WFLP aims at bringing together researchers interested in functional programming, (constraint) logic programming, as well as the integration of the two paradigms. It promotes the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications and combinations of high-level, declarative programming languages and related areas. The previous editions of this workshop were: WFLP 2010 (Madrid, Spain), WFLP 2009 (Brasilia, Brazil), WFLP 2008 (Siena, Italy), WFLP 2007 (Paris, France), WFLP 2006 (Madrid, Spain), WCFLP 2005 (Tallinn, Estonia), WFLP 2004 (Aachen, Germany), WFLP 2003 (Valencia, Spain), WFLP 2002 (Grado, Italy), WFLP 2001 (Kiel, Germany), WFLP 2000 (Benicassim, Spain), WFLP 1999 (Grenoble, France), WFLP 1998 (Bad Honnef, Germany), WFLP 1997 (Schwarzenberg, Germany), WFLP 1996 (Marburg, Germany), WFLP 1995 (Schwarzenberg, Germany), WFLP 1994 (Schwarzenberg, Germany), WFLP 1993 (Rattenberg, Germany), and WFLP 1992 (Karlsruhe, Germany). Since its 2009 edition, the WFLP proceedings have been published by Springer in its *Lecture Notes in Computer Science* series, as volumes 5979 and 6559, respectively.

Each submission of WFLP 2011 was peer-reviewed by at least three Program Committee members with the help of some external experts. The Program Committee meeting was conducted electronically in May 2011 with the help of the conference management system EasyChair. After careful discussions, the Program Committee selected ten submissions for presentation at the workshop. Nine of them were considered mature enough to be published in these proceedings.

On behalf of the Program Committee, I would like to thank all researchers, who gave talks at the workshop, contributed to the proceedings, and submitted papers to WFLP 2011. As Program Chair, I would also like to thank all members of the Program Committee and the external reviewers for their careful work. Moreover, I am pleased to acknowledge the valuable assistance of the conference management system EasyChair and to thank its developer Andrei Voronkov for providing it. Finally, all the participants of WFLP 2011 and I are deeply indebted to Peter Schneider-Kamp and his team, who organized the Odense Summer on Logic and Programming, which besides WFLP comprised the 21st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2011), the 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2011), and the 4th International Workshop on Approaches and Applications of Inductive Programming (AAIP 2011). All the local organizers did a wonderful job and provided invaluable support in the preparation and organization of the overall event.

# Organization

## Program Chair

Herbert Kuchen

University of Münster, Germany

## Program Committee

María Alpuente

Universidad Politécnica de Valencia, Spain

Sergio Antoy

Portland State University, USA

Rafael Caballero-Roldán

Universidad Complutense, Madrid, Spain

Olaf Chitil

University of Kent, UK

Rachid Echahed

Institut IMAG, Laboratoire Leibniz, France

Santiago Escobar

Universidad Politécnica de Valencia, Spain

Moreno Falaschi

University of Siena, Italy

Sebastian Fischer

National Institute of Informatics, Tokyo, Japan

Michael Hanus

Christian-Albrechts-Universität Kiel, Germany

Julio Mariño y Carballo

Universidad Politécnica de Madrid, Spain

Janis Voigtländer

Universität Bonn, Germany

## Additional Reviewers

Demis Ballis

Raúl Gutierrez

Pablo Nogueira

## Local Organization Chair

Peter Schneider-Kamp

University of Southern Denmark, Odense,  
Denmark

# Table of Contents

## Functional Logic Programming

KiCS2: A New Compiler from Curry to Haskell . . . . .	1
<i>Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck</i>	
New Functional Logic Design Patterns . . . . .	19
<i>Sergio Antoy and Michael Hanus</i>	
XQuery in the Functional-Logic Language Toy . . . . .	35
<i>Jesus M. Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez</i>	

## Functional Programming

Size Invariant and Ranking Function Synthesis in a Functional Language . . . . .	52
<i>Ricardo Peña and Agustin D. Delgado-Muñoz</i>	
Memoizing a Monadic Mixin DSL . . . . .	68
<i>Pieter Wuille, Tom Schrijvers, Horst Samulowitz, Guido Tack, and Peter Stuckey</i>	
A Functional Approach to Worst-Case Execution Time Analysis . . . . .	86
<i>Vítor Rodrigues, Mário Florido, and Simão Melo de Sousa</i>	
Building a Faceted Browser in CouchDB Using Views on Views and Erlang Metaprogramming . . . . .	104
<i>Claus Zinn</i>	

## Integration of Constraint Logic and Object-Oriented Programming

Logic Java: Combining Object-Oriented and Logic Programming . . . . .	122
<i>Tim A. Majchrzak and Herbert Kuchen</i>	

## Term Rewriting

On Proving Termination of Constrained Term Rewrite Systems by Eliminating Edges from Dependency Graphs . . . . .	138
<i>Tsubasa Sakata, Naoki Nishida, and Toshiki Sakabe</i>	

Author Index . . . . .	157
------------------------	-----

# KiCS2: A New Compiler from Curry to Haskell

Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany  
{bbr,mh,bjp,fre}@informatik.uni-kiel.de

**Abstract.** In this paper we present our first steps towards a new system to compile functional logic programs of the source language Curry into purely functional Haskell programs. Our implementation is based on the idea to represent non-deterministic results as values of the data types corresponding to the results. This enables the application of various search strategies to extract values from the search space. We show by several benchmarks that our implementation can compete with or outperform other existing implementations of Curry.

## 1 Introduction

Functional logic languages integrate the most important features of functional and logic languages (see [8,24] for recent surveys). In particular, they combine higher-order functions and demand-driven evaluation from functional programming with logic programming features like non-deterministic search and computing with partial information (logic variables). The combination of these features has led to new design patterns [6] and better abstractions for application programming, e.g., as shown for programming with databases [14,18], GUI programming [21], web programming [22,23,26], or string parsing [17].

The implementation of functional logic languages is challenging since a reasonable implementation has to support the operational features mentioned above. One possible approach is the design of new abstract machines appropriately supporting these operational features and implementing them in some (typically, imperative) language, like C [32] or Java [9,27]. Another approach is the reuse of already existing implementations of some of these features by translating functional logic programs into either logic or functional languages. For instance, if one compiles into Prolog, one can reuse the existing backtracking implementation for non-deterministic search as well as logic variables and unification for computing with partial information. However, one has to implement demand-driven evaluation and higher-order functions [5]. A disadvantage of this approach is the commitment to a fixed search strategy (backtracking).

If one compiles into a non-strict functional language like Haskell, one can reuse the implementation of lazy evaluation and higher-order functions, but one has to implement non-deterministic evaluations [13,15]. Although Haskell offers list comprehensions to model backtracking [36], this cannot be exploited due to the specific semantical requirements of the combination of non-strict and non-deterministic operations [20]. Thus, additional implementation efforts are necessary like implementation of shared non-deterministic computations [19].



Nevertheless, the translation of functional logic languages into other high-level languages is attractive: it limits the implementation efforts compared to an implementation from scratch and one can exploit the existing implementation technologies, provided that the efforts to implement the missing features are reasonable.

In this paper we describe an implementation that is based on the latter principle. We present a method to compile programs written in the functional logic language Curry [28] into Haskell programs based on the ideas shown in [12]. The difficulty of such an implementation is the fact that non-deterministic results can occur in any place of a computation. Thus, one cannot separate logic computations by the use of list comprehensions [36], as the outcome of any operation could be potentially non-deterministic, i.e., it might have more than one result value. We solve this problem by an explicit representation of non-deterministic values, i.e., we extend each data type by another constructor to represent the choice between several values. This idea is also the basis of the Curry implementation KiCS [15,16]. However, KiCS is based on unsafe features of Haskell that inhibit the use of optimizations provided by Haskell compilers like GHC<sup>1</sup>. In contrast, our implementation, called KiCS2, avoids such unsafe features. In addition, we also support more flexible search strategies and new features to encapsulate non-deterministic computations (which are not described in detail in this paper due to lack of space).

The general objective of our approach is the support of flexible strategies to explore the search space resulting from non-deterministic computations. In contrast to Prolog-based implementations that use backtracking and, therefore, are incomplete, we also want to support complete strategies like breadth-first search, iterative deepening or parallel search (in order to exploit multi-core architectures). We achieve this goal by an explicit representation of the search space as data that can be traversed by various operations. Moreover, purely deterministic computations are implemented as purely functional programs so that they are executed with almost the same efficiency as their purely functional counterparts.

In the next section, we sketch the source language Curry and introduce a normalized form of Curry programs that is the basis of our translation scheme. Section 3 presents the basic ideas of this translation scheme. Benchmarks of our initial implementation of this scheme are presented in Section 4. Further features of our system are sketched in Section 5 before we conclude in Section 6.

## 2 Curry Programs

The syntax of the functional logic language Curry [28] is close to Haskell [35]. In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. In contrast to functional programming and similarly to logic programming, operations can be defined by overlapping rules so that they might yield more than one result on the same input. Such operations are also called *non-deterministic*. For instance, Curry offers a *choice* operation that is predefined by the following rules:

```
x ? _ = x
_ ? y = y
```

<sup>1</sup> <http://www.haskell.org/ghc/>

Thus, we can define a non-deterministic operation `aBool` by

```
aBool = True ? False
```

so that the expression “`aBool`” has two values: `True` and `False`.

If non-deterministic operations are used as arguments in other operations, a semantic ambiguity might occur. Consider the operations

```
xor True  False = True
xor True  True  = False
xor False x      = x

xorSelf x = xor x x
```

and the expression “`xorSelf aBool`”. If we interpret this program as a term rewriting system, we could have the reduction

```
xorSelf aBool → xor aBool aBool → xor True aBool
               → xor True False → True
```

leading to the unintended result `True`. Note that this result cannot be obtained if we use a strict strategy where arguments are evaluated before the operation calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, González-Moreno et al. [20] proposed the rewriting logic CRWL as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [29] where values of the arguments of an operation are determined before the operation is evaluated. Note that this does not necessarily require an eager evaluation of arguments. Actually, [131] define lazy evaluation strategies for functional logic programs with call-time choice semantics where actual arguments passed to operations are shared. Hence, we can evaluate the expression above lazily, provided that all occurrences of `aBool` are shared so that all of them reduce either to `True` or to `False`. The requirements of the call-time choice semantics are the reason why it is not simply possible to use list comprehensions or non-determinism monads for a straightforward implementation of functional logic programs in Haskell [19].

Due to these considerations, an implementation of Curry has to support lazy evaluation where operations can have multiple results and unevaluated arguments must be shared. This is a complex task, especially if we try to implement it directly on the level of source programs. Therefore, we perform some simplifications on programs before the target code is generated.

First of all, we assume that our programs *do not contain logic variables*. This assumption can be made since it has been shown [7] that logic variables can be replaced by non-deterministic “generators”, i.e., operations that evaluate to all possible values of the type of the logic variable. For instance, a Boolean logic variable can be replaced by the generator `aBool` defined above.

Furthermore, we discuss our translation scheme only for *first-order* programs for the sake of simplicity. However, our implementation also supports higher-order features (see Section 4) by exploiting the corresponding features of Haskell.

$e ::= x$	$x$ is a variable
$c(e_1, \dots, e_n)$	$c$ is an $n$ -ary constructor symbol
$f(e_1, \dots, e_n)$	$f$ is an $n$ -ary function symbol
$e_1 ? e_2$	choice
$D ::= f(x_1, \dots, x_n) = e$	$n$ -ary function $f$ with a single rule
$f(c(y_1, \dots, y_m), x_2, \dots, x_n) = e$	matching rule for $n$ -ary function $f$
	$c$ is an $m$ -ary constructor symbol
$P ::= D_1 \dots D_k$	

**Fig. 1.** Uniform programs ( $e$ : expressions,  $D$ : definitions,  $P$ : programs)

Finally, we assume that the pattern matching strategy is explicitly encoded in individual matching functions. In contrast to [1], where the pattern matching strategy is encoded in case expressions, we assume that each case expression is transformed into a new operation in order to avoid complications arising from the translation of nested case expressions. Thus, we assume that all programs are *uniform* according to the definition in Fig. 1.<sup>2</sup> There, the variables in the left-hand sides of each rule are pairwise different, and the constructors in the left-hand sides of the matching rules of each function are pairwise different. Uniform programs have a simple form of pattern matching: either a function is defined by a single rule without pattern matching, or it is defined by rules with only one constructor in the left-hand side of each rule, and in the same argument for all rules.<sup>3</sup> For instance, the operation `xor` defined above can be transformed into the following uniform program:

```
xor True   x = xor' x
xor False  x = x
xor' False = True
xor' True  = False
```

In particular, there are no overlapping rules for functions (except for the choice operation “?” which is considered as predefined). Antoy [3] showed that each functional logic program, i.e., each constructor-based conditional term rewriting system, can be translated into an equivalent unconditional term rewriting system without overlapping rules but containing choices in the right-hand sides, also called LOIS (limited overlapping inductively sequential) system. Furthermore, Braßel [11] showed the semantical equivalence of narrowing computations in LOIS systems and rewriting computations in uniform programs. Due to these results, uniform programs are a reasonable intermediate language for our translation into Haskell which will be presented in the following.

<sup>2</sup> A variant of uniform programs has been considered in [33] to define lazy narrowing strategies for functional logic programs. Although the motivations are similar, our notion of uniform programs is more restrictive since we allow only a single non-variable argument in each left-hand side of a rule. Uniform programs have also been applied in [37] to define a denotational analysis of functional logic programs.

<sup>3</sup> For simplicity, we require in Fig. 1 that the matching argument is always the first one, but one can also choose any other argument.

### 3 Compilation to Haskell: The Basics

#### 3.1 Representing Non-deterministic Computations

As mentioned above, our implementation is based on the explicit representation of non-deterministic results in a data structure. This can easily be achieved by adding a constructor to each data type to represent a choice between two values. For instance, one can redefine the data type for Boolean values as follows:

```
data Bool = False | True | Choice Bool Bool
```

Thus, we can implement the non-deterministic operation `aBool` defined in Section 2 as:

```
aBool = Choice True False
```

If operations can deliver non-deterministic values, we have to extend the rules for operations defined by pattern matching so that they do not fail on non-deterministic argument values. Instead, they move the non-deterministic choice one level above, i.e., a choice in some argument leads to a choice in any result of this operation (this is also called a “pull-tab” step in [2]). For instance, the rules of the uniform operation `xor` shown above are extended as follows:

```
xor True      x = xor' x
xor False     x = x
xor (Choice x1 x2) x = Choice (xor x1 x) (xor x2 x)

xor' False    = True
xor' True     = False
xor' (Choice x1 x2) = Choice (xor' x1) (xor' x2)
```

The operation `xorSelf` is not defined by a pattern matching rule and, thus, need not be changed. If we evaluate the expression “`xorSelf aBool`”, we get the result

```
Choice (Choice False True) (Choice True False)
```

How can we interpret this result? In principle, the choices represent different possible values. Thus, if we want to show the different values of an expression (which is usually the task of a top-level “read-eval-print” loop), we enumerate all values contained in the choices. These are `False`, `True`, `True`, and `False` in the result above. Unfortunately, this does not conform to the call-time choice semantics discussed in Section 2 which excludes a value like `True`. The call-time choice semantics requires that the choice of a value made for the initial expression `aBool` should be consistent in the entire computation. For instance, if we select the value `False` for the expression `aBool`, this selection should be made at all other places where this expression might have been copied during the computation. However, our initial implementation duplicates the initially single `Choice` into finally three occurrences of `Choice`.

We can correct this unintended behavior of our implementation by identifying different `Choice` occurrences that are duplicates of some single `Choice`. This can be easily done by attaching a unique identifier, e.g., a number, to each choice:

```
type ID = Integer
data Bool = False | True | Choice ID Bool Bool
```

Furthermore, we modify the `Choice` pattern rules so that the identifiers will be kept, e.g.,

```
xor (Choice i x1 x2) x = Choice i (xor x1 x) (xor x2 x)
```

If we evaluate the expression “`xorSelf aBool`” and assign the number 1 to the choice of `aBool`, we obtain the result

```
Choice 1 (Choice 1 False True) (Choice 1 True False)
```

When we show the values contained in this result, we have to make *consistent* selections in choices with same identifiers. Thus, if we select the left branch as the value of the outermost `Choice`, we also have to select the left branch in the selected argument (`Choice 1 False True`) so that only the value `False` is possible here. Similarly, if we select the right branch as the value of the outermost `Choice`, we also have to select the right branch in its selected argument (`Choice 1 True False`) which yields the sole value `False`.

Note that each `Choice` occurring for the first time in a computation has to get its own unique identifier. For instance, if we evaluate the expression “`xor aBool aBool`”, the two occurrences of `aBool` assign different identifiers to their `Choice` constructor (e.g., 1 for the left and 2 for the right `aBool` argument) so that this evaluates to

```
Choice 1 (Choice 2 False True) (Choice 2 True False)
```

Here we can make different selections for the outer and inner `Choice` constructors so that this non-deterministic result represents four values.

To summarize, our implementation is based on the following principles:

1. Each non-deterministic choice is represented by a `Choice` constructor with a unique identifier.
2. When matching a `Choice` constructor, the choice is moved to the result of this operation with the same identifier, i.e., a non-deterministic argument yields non-deterministic results for each of the argument’s values.
3. Each choice introduced in a computation is supplied with its own unique identifier.

The latter principle requires the creation of fresh identifiers during a computation—a non-trivial problem in functional languages. One possibility is the use of a global counter that is accessed by unsafe features whenever a new identifier is required. Unfortunately, unsafe features inhibit the use of optimization techniques developed for purely functional programs and make the application of advanced evaluation and search strategies (e.g., parallel strategies) more complex. Therefore, we avoid unsafe features in our implementation. Instead, we thread some global information through our program in order to supply fresh references at any point of a computation. For this purpose, we assume a type `IDSupply` with operations

```
initSupply :: IO IDSupply
thisID     :: IDSupply → ID
leftSupply :: IDSupply → IDSupply
rightSupply :: IDSupply → IDSupply
```

and add a new argument of type `IDSupply` to each operation of the source program, i.e., a Curry operation of type

```
f :: τ1 → ... → τn → τ
```

is translated into a Haskell function of type

```
f :: τ1 → ... → τn → IDSupply → τ
```

Conceptually, one can consider `IDSupply` as an infinite set of identifiers that is created at the beginning of an evaluation by the operation `initSupply`. The operation `thisID` takes some identifier from this set, and `leftSupply` and `rightSupply` split this set into two disjoint subsets without the identifier obtained by `thisID`. The split operations `leftSupply` and `rightSupply` are used when an operation calls two<sup>4</sup> other operations in the right-hand side of a rule. In this case, the called operations must be supplied with their individual disjoint identifier supplies. For instance, the operation `main` defined by

```
main :: Bool
main = xorSelf aBool
```

is translated into

```
main :: IDSupply → Bool
main s = xorSelf (aBool (leftSupply s)) (rightSupply s)
```

Any choice in the right-hand side of a rule gets its own identifier by the operation `thisID`, as in

```
aBool s = Choice (thisID s) True False
```

The type `IDSupply` can be implemented in various ways. The simplest implementation uses unbounded integers:

```
type IDSupply = Integer
initSupply    = return 1
thisID       n = n
leftSupply   n = 2*n
rightSupply  n = 2*n+1
```

There are other more sophisticated implementations available [10]. Actually, our compilation system is parameterized over different implementations of `IDSupply` in order to perform some experiments and choose the most appropriate for a given application. Each implementation must ensure that, if  $s$  is a value of type `IDSupply`, then `thisID(o1(... (on s) ...))` and `thisID(o'1(... (o'm s) ...))` are different identifiers provided that  $o_i, o'_j \in \{\text{leftSupply}, \text{rightSupply}\}$  and  $o_1 \cdots o_n \neq o'_1 \cdots o'_m$ .

### 3.2 The Basic Translation Scheme

Functional logic computations can also fail, e.g., due to partially defined operations. Computing with failures is a typical programming technique and provides for specific programming patterns [6]. Hence, in contrast to functional programming, a failing computation should not abort the complete evaluation but it should be considered as some part of a computation that does not produce a meaningful result. In order to implement this behavior, we extend each data type by a further constructor `Fail` and complete

<sup>4</sup> The extension to more than two is straightforward.

each operation containing matching rules by a final rule that matches everything and returns the value `Fail`. For instance, consider the definition of lists

```
data List a = Nil | Cons a (List a)
```

and an operation to extract the first element of a non-empty list:

```
head :: List a → a
head (Cons x xs) = x
```

The type definition is extended as follows<sup>5</sup>

```
data List a = Nil | Cons a (List a) | Choice (List a) (List a) | Fail
```

The operation `head` is extended by an identifier supply and further matching rules:

```
head :: List a → IDSupply → a
head (Cons x xs)      s = x
head (Choice i x1 x2) s = Choice i (head x1 s) (head x2 s)
head _                s = Fail
```

Note that the final rule returns `Fail` if `head` is applied to the empty list as well as if the matching argument is already a failed computation, i.e., it also propagates failures.

As already discussed above, an occurrence of “?” in the right-hand side is translated into a `Choice` supplied with a fresh identifier by the operation `thisID`. In order to ensure that each occurrence of “?” in the source program get its own identifier, all choices and all operations in the right-hand side of a rule get their own identifier supplies via appropriate applications of `leftSupply` and `rightSupply` to the supply of the defined operation. For instance, a rule like

```
main2 = xor aBool (False ? True)
```

is translated into

```
main2 s = let s1 = leftSupply s
            s2 = rightSupply s
            s3 = leftSupply s2
            s4 = rightSupply s2
          in xor (aBool s3) (Choice (thisID s4) False True) s1
```

An obvious optimization, performed by our compiler, is a *determinism analysis*. If an operation does not call, neither directly nor indirectly through other operations, the choice operation “?”, then it is not necessary to pass a supply for identifiers. In this case, the `IDSupply` argument can be omitted so that the generated code is nearly identical to a corresponding functional program (apart from the additional rules to match the constructors `Choice` and `Fail`).

As mentioned in Section 2, our compiler translates occurrences of logic variables into generators. Since these generators are standard non-deterministic operations, they are translated like any other operation. For instance, the operation `aBool` is a generator for Boolean values and its translation into Haskell has been presented above.

<sup>5</sup> Actually, our compiler performs some renamings to avoid conflicts with predefined Haskell entities and introduces type classes to resolve overloaded symbols like `Choice` and `Fail`.

A more detailed discussion of this translation scheme can be found in the original proposal [12]. The correctness of this transformation from non-deterministic source programs into deterministic target programs is formally shown in [11].

### 3.3 Extracting Values

So far, our generated operations compute all the non-deterministic values of an expression represented by a structure containing `Choice` constructors. In order to extract the various values from this structure, we have to define operations that compute all possible choices in some order where the choice identifiers are taken into account. To provide a common interface for such operations, we introduce a data type to represent the general outcome of a computation,

```
data Try a = Val a | Choice ID a a | Fail
```

together with an auxiliary operation<sup>6</sup>

```
try :: a → Try a
try (Choice i x y) = Choice i x y
try Fail           = Fail
try x              = Val x
```

In order to take the identity of choices into account when extracting values, one has to remember which choice (e.g., left or right branch) has been made for some particular choice. Therefore, we introduce the type

```
data Choice = NoChoice | ChooseLeft | ChooseRight
```

where `NoChoice` represents the fact that a choice has not yet been made. Furthermore, we need operations to lookup the current choice for a given identifier or change its choice:

```
lookupChoice :: ID → IO Choice
setChoice    :: ID → Choice → IO ()
```

In Haskell, there are different possibilities to implement a mapping from choice identifiers to some value of type `Choice`. Our implementation supports various options together with different implementations of `IDSupply`. For instance, a simple but efficient implementation can be obtained by using updatable values, i.e., the Haskell type `IORef`. In this case, choice identifiers are memory cells instead of integers:

```
newtype ID = ID (IORef Choice)
```

Consequently, the implementation of `IDSupply` requires an infinite set of memory cells which can be represented as a tree structure:

```
data IDSupply = IDSupply ID IDSupply IDSupply
thisID       (IDSupply r _ _) = r
leftSupply   (IDSupply _ s _) = s
rightSupply  (IDSupply _ _ s) = s
```

---

<sup>6</sup> Note that the operation `try` is not really polymorphic but overloaded for each data type and, therefore, defined in instances of some type class.



The infinite tree of memory cells (with initial value `NoChoice`) can be constructed as follows, where `unsafeInterleaveIO` is used to construct the tree on demand:

```
initSupply = getIDTree
getIDTree = do s1 <- unsafeInterleaveIO getIDTree
              s2 <- unsafeInterleaveIO getIDTree
              r  <- unsafeInterleaveIO (newIORef NoChoice)
              return (IDSupply (ID r) s1 s2)
```

Using memory cells, the implementation of the lookup and set operations is straightforward:

```
lookupChoice (ID ref) = readIORef ref
setChoice (ID ref) c = writeIORef ref c
```

Now we can print all values contained in a choice structure in a depth-first manner by the following operation:

```
printValsDFS :: Try a → IO ()
printValsDFS (Val v)      = print v
printValsDFS Fail        = return ()
printValsDFS (Choice i x1 x2) = lookupChoice i >>= choose
  where
    choose ChooseLeft  = printValsDFS (try x1)
    choose ChooseRight = printValsDFS (try x2)
    choose NoChoice    = do newChoice ChooseLeft x1
                           newChoice ChooseRight x2

newChoice ch x = do setChoice i ch
                    printValsDFS (try x)
                    setChoice i NoChoice
```

This operation prints a computed value and ignores failures. If there is some choice, it checks whether a choice for this identifier has already been made (note that the initial value for all identifiers is `NoChoice`). If a choice has been made, it follows this choice. Otherwise, the left choice is made and stored. After printing all the values w.r.t. this choice, the choice is undone (like in backtracking) and the right choice is made and stored.

For instance, to print all values of the expression `main` defined in Section [3.1](#), we evaluate the Haskell expression

```
initSupply >>= \s → printValsDFS (try (main s))
```

Thus, we obtain the output

```
False
False
```

In general, one has to propagate all choices and failures to the top level of a computation before printing the results. Otherwise, the operation `try` applied to an expression like “`Just aBool`” would return a `Val`-structure instead of a `Choice` so that the main operation `printValsDFS` would miss the non-determinism of the result value. Therefore, we have to compute the normal form of the main expression before passing it to the operation `try`. Hence, the result values of `main` are printed by evaluating

```
initSupply >=> \s → printValsDFS (try (id $!! main s))
```

where “ $f \$!! x$ ” denotes the application of the operation  $f$  to the normal form of its argument  $x$ . This has the effect that a choice or failure occurring somewhere in a computation will be moved (by the operation “ $\$!!$ ”) to the root of the main expression so that the corresponding search strategy can process it. This ensures that, after the computation to a normal form, an expression without a `Choice` or `Fail` at the root is a value, i.e., it does not contain a `Choice` or `Fail`.

Of course, printing all values via depth-first search is only one option which is not sufficient in case of infinite search spaces. For instance, one can easily define an operation that prints only the first solution. Due to the lazy evaluation strategy of Haskell, such an operation can also be applied to infinite choice structures. In order to abstract from these different printing options, our implementation contains a more general approach by translating choice structures into monadic structures w.r.t. various strategies (depth-first search, breadth-first search, iterative deepening, parallel search). This allows for an independent processing of the resulting monadic structures, e.g., by an interactive loop where the user can request the individual values.

## 4 Benchmarks

In this section we evaluate our compiler by comparing the efficiency of the generated Haskell programs to various other systems, in particular, other implementations of Curry. For our comparison with other Curry implementations, we consider PAKCS [25] (Version 1.9.2) which compiles Curry into Prolog [5] (based on SICStus-Prolog 4.1.2, a SWI-Prolog 5.10 back end is also available but much slower). PAKCS has been used for a number of practical applications of Curry. Another mature implementation we consider is MCC [32] (Version 0.9.10) which compiles Curry into C. MonC [13] is a compiler from Curry into Haskell. It is based on a monadic representation of non-deterministic computations where sharing is explicitly managed by the technique proposed in [19]. Since this compiler is in an experimental state, we could not execute all benchmarks with MonC (these are marked by “n/a”).

The functional logic language TOY [30] has many similarities to Curry and the TOY system compiles TOY programs into Prolog programs. However, we have not included a comparison in this paper since [5] contains benchmarks showing that the implementation of sharing used in PAKCS produces more efficient programs.

Our compiler has been executed with the Glasgow Haskell Compiler (GHC 6.12.3, option -O2). All benchmarks were executed on a Linux machine running Debian 5.0.7 with an Intel Core 2 Duo (3.0GHz) processor. The timings were performed with the time command measuring the execution time (in seconds) of a compiled executable for each benchmark as a mean of three runs. “oom” denotes a memory overflow in a computation.

The first collection of benchmarks<sup>7</sup> (Fig. 2) are purely first-order functional programs. The Prolog (SICStus, SWI) and Haskell (GHC) programs have been rewritten

<sup>7</sup> All benchmarks are available at <http://www-ps.informatik.uni-kiel.de/kics2/benchmarks/>

System	ReverseUser	Reverse	Tak	TakPeano
KiCS2	0.12	0.12	0.21	0.79
PAKCS	2.05	1.88	39.80	62.43
MCC	0.43	0.47	1.21	5.49
MonC	23.39	22.00	20.37	oom
GHC	0.12	0.12	0.04	0.49
SICStus	0.39	0.29	0.49	5.20
SWI	1.63	1.39	1.84	11.66

**Fig. 2.** Benchmarks: first-order functional programs

according to the Curry formulation. “ReverseUser” is the naive reverse program applied to a list of 4096 elements, where all data (lists, numbers) are user-defined. “Reverse” is the same but with built-in lists. “Tak” is a highly recursive function on naturals [34] applied to arguments (27,16,8) and “TakPeano” is the same but with user-defined natural numbers in Peano representation. Note that the Prolog programs use a strict evaluation strategy in contrast to all others. Thus, the difference between PAKCS and SICStus shows the overhead to implement lazy evaluation in Prolog.

One can deduce from these results that one of the initial goals for this compiler is satisfied, since functional Curry programs are executed almost with the same speed as their Haskell equivalents. An overhead is visible if one uses built-in numbers (due to the potential non-deterministic values, KiCS2 cannot directly map operations on numbers into the Haskell primitives) where GHC can apply specific optimizations.

System	ReverseHO	Primes	PrimesPeano	Queens	QueensUser
KiCS2	oom	1.22	0.30	10.02	13.08
KiCS2HO	0.24	0.09	0.27	0.65	0.73
PAKCS	7.97	14.52	23.08	81.72	81.98
MCC	0.27	0.32	1.77	3.25	3.62
MonC	oom	16.74	oom	oom	oom
GHC	0.24	0.06	0.22	0.06	0.11

**Fig. 3.** Benchmarks: higher-order functional programs

The next collection of benchmarks (Fig. 3) considers higher-order functional programs so that we drop the comparison to first-order Prolog systems. “ReverseHO” reverses a list with one million elements in linear time using higher-order functions like `foldl` and `flip`. “Primes” computes the 2000th prime number via the sieve of Eratosthenes using higher-order functions, and “PrimesPeano” computes the 256th prime number but with Peano numbers and user-defined lists. Finally, “Queens” (and “QueensUser” with user-defined lists) computes the number of safe positions of 11 queens on a  $11 \times 11$  chess board.

As discussed above, our compiler performs an optimization when all operations are deterministic. However, in the case of higher-order functions, this determinism optimization cannot be performed since any operation, i.e., also a non-deterministic

operation, can be passed as an argument. As shown in the first line of this table, this considerably reduces the overall performance. To improve this situation, our compiler generates two versions of a higher-order function: a general version applicable to any argument and a specialized version where all higher-order arguments are assumed to be deterministic operations. Moreover, we implemented a program analysis to approximate those operations that call higher-order functions with deterministic operations so that their specialized versions are used. The result of this improvement is shown as “KiCS2HO” and demonstrates its usefulness. Therefore, it is always used in the subsequent benchmarks.

System	PermSort	PermSortPeano	Last	RegExp
KiCS2HO	2.83	3.68	0.14	0.49
PAKCS	26.96	67.11	2.61	12.70
MCC	1.46	5.74	0.09	0.57
MonC	48.15	916.61	n/a	n/a

**Fig. 4.** Benchmarks: non-deterministic functional logic programs

To evaluate the efficiency of non-deterministic computations (Fig. 4), we sort a list containing 15 elements by enumerating all permutations and selecting the sorted ones (“PermSort” and “PermSortPeano” for Peano numbers), compute the last element  $x$  of a list  $xs$  containing 100,000 elements by solving the equation “ $ys++[x] == xs$ ” (the implementation of unification and variable bindings require some additional machinery that is sketched in Section 5.3), and match a regular expression in a string of length 200,000 following the non-deterministic specification of `grep` shown in [8]. The results show that our high-level implementation is not far from the efficiency of MCC, and it is superior to PAKCS which exploits Prolog features like backtracking, logic variables and unification for these benchmarks.

Since our implementation represents non-deterministic values as Haskell data structures, we get, in contrast to most other implementations of Curry, one interesting improvement for free: deterministic subcomputations are shared even if they occur in different non-deterministic computations. To show this effect of our implementation, consider the non-deterministic sort operation `psort` (permutation sort) and the infinite list of all prime numbers `primes`, as used in the previous benchmarks, and the following definitions:

```
goal1 = [primes!!1003, primes!!1002, primes!!1001, primes!!1000]
goal2 = psort [7949,7937,7933,7927]
goal3 = psort [primes!!1003, primes!!1002, primes!!1001, primes!!1000]
```

In principle, one would expect that the sum of the execution times of `goal1` and `goal2` is equal to the time to execute `goal3`. However, implementations based on backtracking evaluate the primes occurring in `goal3` multiple times, as can be seen by the run times for PAKCS and MCC shown in Fig 5.

System	goal1	goal2	goal3
KICS2HO	0.34	0.00	0.34
PAKCS	14.90	0.02	153.65
MCC	0.33	0.00	3.46

Fig. 5. Benchmarks: sharing over non-determinism

## 5 Further Features

In this section we sketch some additional features of our implementation. Due to lack of space, we cannot discuss them in detail.

### 5.1 Search Strategies

Due to the fact that we represent non-deterministic results in a data structure rather than as a computation as in implementations based on backtracking, we can provide different methods to explore the search space containing the different result values. We have already seen in Section 3.3 how this search space can be explored to print all values in depth-first order. Apart from this simple approach, our implementation contains various strategies (depth-first, breadth-first, iterative deepening, parallel search) to transform a choice structure into a list of results that can be printed in different ways (e.g., all solutions, only the first solution, or one after another by user requests). Actually, the user can set options to select the search strategy and the printing method.

Method	PermSort	PermSortPeano	NDNums
printValsDFS	2.82	3.66	$\infty$
depth-first search	5.33	6.09	$\infty$
breadth-first search	26.16	29.25	34.00
iterative deepening	9.16	10.91	0.16

Fig. 6. Benchmarks: comparing different search strategies

In order to compare the various search strategies, Fig. 6 contains some corresponding benchmarks. “PermSort” and “PermSortPeano” are the programs discussed in Fig. 4 and “NDNums” is the program

```
f n = f (n+1) ? n
```

where we look for the first solution of “ $f\ 0 == 25000$ ” (obviously, depth-first search strategies do not terminate on this equation). All strategies except for the “direct print” method `printValsDFS` translate choice structures into monadic list structures in order to print them according to the user options. The benchmarks show that the overhead of this transformation is acceptable so that this more flexible approach is the default one.

We also made initial benchmarks with a parallel strategy where non-deterministic choices are explored via GHC’s `par` construct. For the permutation sort we obtained a speedup of 1.7 when executing the same program on two processors, but no essential

speedup is obtained for more than two processors. Better results require a careful analysis of the synchronization caused by the global structure to manage the state of choices. This is a topic for future work.

## 5.2 Encapsulated Search

In addition to the different search strategies to evaluate top-level expressions, our system also contains a primitive operation

```
searchTree :: a → IO (SearchTree a)
```

to translate the search space caused by the evaluation of its argument into a tree structure of the form

```
data SearchTree a = Value a | Fail | Or (SearchTree a) (SearchTree a)
```

With this primitive, the programmer can define its own search strategy or collect all non-deterministic values into a list structure for further processing [15].

## 5.3 Logic Variables and Unification

Although our implementation is based on eliminating all logic variables from the source program by introducing generators, many functional logic programs contain equational constraints (e.g., compare the example “Last” of Fig. 4) to put conditions on computed results. Solving such conditions by generating all values is not always a reasonable approach. For instance, if  $x_s$  and  $y_s$  are free variables of type  $[Bool]$ , the equational constraint “ $x_s = y_s$ ” has an infinite number of solutions. Instead of enumerating all these solutions, it is preferable to delay this enumeration but remember the condition that both  $x_s$  and  $y_s$  must always be evaluated to the same value. This demands for extending the representation of non-deterministic values by the possibility to add equational constraints between different choice identifiers. Due to lack of space, we have to omit the detailed description of this extension. However, it should be noted that the examples “Last” and “RegExp” of Fig. 4 show that unification can be supported with a reasonable efficiency.

## 6 Conclusions and Related Work

We have presented a new system to compile functional logic programs into purely functional programs. In order to be consistent with the call-time choice semantics of functional logic languages like Curry or TOY, we represent non-deterministic values in choice structures where each choice has an identification. Values for such choice identifiers are passed through non-deterministic operations so that fresh identifiers are available when a new choice needs to be created. The theoretical justification of this implementation technique is provided in [11]. Apart from the parser, where we reused an existing one implemented in Haskell, the compiler is completely written in Curry.

Due to the representation of non-deterministic values as data, our system easily supports various search strategies in contrast to Prolog-based implementations. Since we compile Curry programs into Haskell, we can exploit the implementation efforts

done for functional programming. Hence, purely functional parts of functional logic programs can be executed with almost the same efficiency as Haskell programs. Our benchmarks show that even the execution of the non-deterministic parts can compete with other implementations of Curry.

In the introduction we already discussed the various efforts to implement functional logic languages, like the construction of abstract machines [9,27,32] and the compilation into Prolog [5] or Haskell [13,15,16]. Our benchmarks show that an efficient implementation by compiling into a functional language depends on carefully handling the sharing of non-deterministic choices. For instance, our previous implementation [13], where sharing is explicitly managed by the monadic techniques proposed in [19], has not satisfied the expectations that came from the benchmarks reported in [19]. Due to these experiences, in our new compiler we use the compilation scheme initially proposed in [12] which produces much faster code, as shown in our benchmarks.

If non-deterministic results are collected in data structures, one has more fine-grained control over non-deterministic steps. For instance, [2] proposes pull-tab steps to move non-determinism from arguments to the result position of a function. Antoy [4] shows that single pull-tab steps are semantics-preserving. Thus, it is not necessary to move each choice to the root of an expression, as done in our implementation, but one could also perform further local computations in the arguments of a choice before moving it up. This might be a reasonable strategy if all non-deterministic values are required but many computations fail. However, the general effects of such refinements need further investigations.

Our implementation has many opportunities for optimization, like better program analyses to approximate purely deterministic computations. We can also exploit advanced developments in the implementation of Haskell, like the parallel evaluation of expressions. These are interesting topics for future work.

## 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. Alqaddoumi, A., Antoy, S., Fischer, S., Reck, F.: The pull-tab transformation. In: *Proc. of the Third International Workshop on Graph Computation Models*, pp. 127–132. Enschede, The Netherlands (2010), <http://gcm-events.org/gcm2010/pages/gcm2010-preproceedings.pdf>
3. Antoy, S.: Constructor-based conditional narrowing. In: *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 199–206. ACM Press, New York (2001)
4. Antoy, S.: On the correctness of the pull-tab transformation. In: *To Appear in Proceedings of the 27th International Conference on Logic Programming, ICLP 2011* (2011)
5. Antoy, S., Hanus, M.: Compiling multi-paradigm declarative programs into Prolog. In: Kirchner, H. (ed.) *FroCos 2000*. LNCS, vol. 1794, pp. 171–185. Springer, Heidelberg (2000)
6. 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)
7. Antoy, S., Hanus, M.: Overlapping rules and logic variables in functional logic programs. In: Etalle, S., Truszczyński, M. (eds.) *ICLP 2006*. LNCS, vol. 4079, pp. 87–101. Springer, Heidelberg (2006)

8. Antoy, S., Hanus, M.: Functional logic programming. *Communications of the ACM* 53(4), 74–85 (2010)
9. Antoy, S., Hanus, M., Liu, J., Tolmach, A.: A virtual machine for functional logic computations. In: Grelck, C., Huch, F., Michaelson, G.J., Trinder, P. (eds.) *IFL 2004*. LNCS, vol. 3474, pp. 108–125. Springer, Heidelberg (2005)
10. Augustsson, L., Rittri, M., Synek, D.: On generating unique names. *Journal of Functional Programming* 4(1), 117–123 (1994)
11. Braßel, B.: Implementing Functional Logic Programs by Translation into Purely Functional Programs. PhD thesis, Christian-Albrechts-Universität zu Kiel (2011)
12. Braßel, B., Fischer, S.: From functional logic programs to purely functional programs preserving laziness. In: *Pre-Proceedings of the 20th Workshop on Implementation and Application of Functional Languages, IFL 2008* (2008)
13. Braßel, B., Fischer, S., Hanus, M., Reck, F.: Transforming functional logic programs into monadic functional programs. In: Mariño, J. (ed.) *WFLP 2010*. LNCS, vol. 6559, pp. 30–47. Springer, Heidelberg (2011)
14. 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)
15. 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)
16. Braßel, B., Huch, F.: The kiel curry system kiCS. In: Seipel, D., Hanus, M., Wolf, A. (eds.) *INAP 2007*. LNCS(LNAI), vol. 5437, pp. 195–205. Springer, Heidelberg (2009)
17. Caballero, R., López-Fraguas, F.J.: A functional-logic perspective of parsing. In: Middeldorp, A. (ed.) *FLOPS 1999*. LNCS, vol. 1722, pp. 85–99. Springer, Heidelberg (1999)
18. Fischer, S.: A functional logic database library. In: *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 54–59. ACM Press, New York (2005)
19. Fischer, S., Kiselyov, O., Shan, C.: Purely functional lazy non-deterministic programming. In: *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pp. 11–22. ACM, New York (2009)
20. González-Moreno, J.C., Hortalá-González, M.T., López-Fraguas, F.J., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming* 40, 47–87 (1999)
21. 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)
22. Hanus, M.: High-level server side web scripting in curry. In: Ramakrishnan, I.V. (ed.) *PADL 2001*. LNCS, vol. 1990, pp. 76–92. Springer, Heidelberg (2001)
23. 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)
24. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) *ICLP 2007*. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
25. 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* (2010), <http://www.informatik.uni-kiel.de/~pakcs/>
26. Hanus, M., Koschnicke, S.: An ER-based framework for declarative web programming. In: Carro, M., Peña, R. (eds.) *PADL 2010*. LNCS, vol. 5937, pp. 201–216. Springer, Heidelberg (2010)
27. Hanus, M., Sadre, R.: An abstract machine for curry and its concurrent implementation in java. *Journal of Functional and Logic Programming* 1999(6) (1999)



28. Hanus, M. (ed.): Curry: An integrated functional logic language, vers. 0.8.2 (2006), <http://www.curry-language.org>
29. Hussmann, H.: Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming* 12, 237–255 (1992)
30. Fraguas, F.J.L., Hernández, J.S.: TOY: A multiparadigm declarative system. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
31. López-Fraguas, F.J., Rodríguez-Hortálá, J., Sánchez-Hernández, J.: A simple rewrite notion for call-time choice semantics. In: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2007), pp. 197–208. ACM Press, New York (2007)
32. Lux, W.: Implementing encapsulated search for a lazy functional logic language. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 100–113. Springer, Heidelberg (1999)
33. Moreno-Navarro, J.J., Kuchen, H., Loogen, R., Rodríguez-Artalejo, M.: Lazy narrowing in a graph machine. In: Kirchner, H., Wechler, W. (eds.) ALP 1990. LNCS, vol. 463, pp. 298–317. Springer, Heidelberg (1990)
34. Partain, W.: The nofib benchmark suite of Haskell programs. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming, pp. 195–202. Springer, Heidelberg (1993)
35. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, Cambridge (2003)
36. Wadler, P.: How to replace failure by a list of successes. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985)
37. Zartmann, F.: Denotational Abstract Interpretation of Functional Logic Programs. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 141–156. Springer, Heidelberg (1997)

# New Functional Logic Design Patterns

Sergio Antoy<sup>1</sup> and Michael Hanus<sup>2</sup>

<sup>1</sup> Computer Science Dept., Portland State University, Oregon, U.S.A.

`antoy@cs.pdx.edu`

<sup>2</sup> Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany

`mh@informatik.uni-kiel.de`

**Abstract.** Patterns distill successful experience in solving common software problems. We introduce a handful of new software design patterns for functional logic languages. Some patterns are motivated by the evolution of the paradigm in the last 10 years. Following usual approaches, for each pattern we propose a name and we describe its intent, applicability, structure, consequences, etc. Our patterns deal with fundamental aspects of the design and implementation of functional logic programs such as function invocation, data structure representation and manipulation, specification-driven implementation, pattern matching, and non-determinism. We present some problems and we show fragments of programs that solve these problems using our patterns. The programming language of our examples is Curry. The complete programs are available on-line.

## 1 Introduction

A *design pattern* is a proven solution to a recurring problem in software design and development. A pattern itself is not primarily code. Rather it is an expression of design decisions affecting the architecture of a software system. A pattern consists of both ideas and recipes for the implementations of these ideas often in a particular language or paradigm. The ideas are reusable, whereas their implementations may have to be customized for each problem. For example, the *Constrained Constructor* pattern [3], expresses the idea of calling a data constructor exclusively indirectly through an intermediate function to avoid undesirable instances of some type. The idea is applicable to a variety of problems, but the code of the intermediate function is dependent on each problem.

Patterns originated from the development of object-oriented software [6] and became both a popular practice and an engineering discipline after [11]. As the landscape of programming languages evolves, patterns are “translated” from one language into another [10,12]. Some patterns are primarily language specific, whereas others are fundamental enough to be largely independent of the language or programming paradigm in which they are coded. For example, the *Adapter* pattern [11], which solves the problem of adapting a service to a client coded for different interface, is language independent. The *Facade* pattern [11], which presents a set of separately coded services as a single unit, depends more on

the modularization features of a language than the language’s paradigm itself. The *Visitor* pattern [11], which enables extending the functionality of a class without modifying the class interface, is critically dependent on features of object orientation, such as overloading and overriding.

Patterns are related to both idioms and pearls. Patterns are more articulated than idioms, which never cross languages boundaries, and less specialized than pearls, which often are language specific. The boundaries of these concepts are somewhat arbitrary. Patterns address general structural problems and therefore we use this name for our concepts.

Patterns for a declarative paradigm—in most cases specifically for a functional logic one—were introduced in [3]. This paper is a follow up. Ten years of active research in functional logic programming have brought new ideas and deeper understanding, and in particular some new features and constructs, such as functional patterns [4] and set functions [5]. Some patterns presented in this paper originates from these developments.

High-level languages are better suited for the implementation of reusable code than imperative languages, see, e.g., parser combinators [7]. Although whenever possible we attempt to provide reusable code, the focus of our presentation is on the reusability of design and architecture which are more general than the code itself. Our primary emphasis is not on efficiency, but on clarity and simplicity of design and ease of understanding and maintenance. Interestingly enough, one of our patterns is concerned with moving from the primary emphasis to more efficient code. Our presentation of a pattern follows the usual (metapattern) approaches that provide, e.g., name, intent, applicability, structure, consequences, etc. Some typical elements, such as “known uses,” are sparse or missing because functional logic programming is a still relatively young paradigm. Work on patterns for this paradigm is slowly emerging.

Section 2 briefly recalls some principles of functional logic programming and the programming language Curry which we use to present the examples. Section 3 presents a small catalog of functional logic patterns together with motivating problems and implementation fragments. Section 4 concludes the paper.

## 2 Functional Logic Programming and Curry

A Curry program is a set of functions and data type definitions in Haskell-like syntax [26]. Data type definitions have the same semantics as Haskell. A function is defined by conditional rewrite rules of the form:

$$f\ t_1 \dots t_n \mid c = e \quad \text{where } vs \text{ free} \tag{1}$$

Type variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of  $f$  to  $e$  is denoted by juxtaposition (“ $f\ e$ ”).

In addition to Haskell, Curry offers two main features for logic programming: logical (free) variables and non-deterministic functions. Logical variables are declared by a “free” clause as shown above, can occur in conditions and/or

right-hand sides of defining rules, and are instantiated by narrowing [112], a computation similar to resolution, but applicable to functions of any type rather than predicates only. Similarly to Haskell, the “**where**” clause is optional and can also contain other local function and/or pattern definitions.

Non-deterministic functions are defined by overlapping rules such as:

```
(?) :: a -> a -> a
x ? y = x
x ? y = y
```

In contrast to Haskell, in which the first matching rule is applied, in Curry all matching (to be more precise, unifiable) rules are applied—non-deterministically. For example,  $0 ? 1$  has two values, 0 and 1. The programmer has no control over which value is selected during an execution, but will typically constrain this value according to the intent of the program. In particular, Curry defines *equational constraints* of the form  $e_1 = e_2$  which are satisfiable if both sides  $e_1$  and  $e_2$  are reducible to unifiable data terms. Furthermore, “ $c_1 \& c_2$ ” denotes the *concurrent conjunction* of the constraints  $c_1$  and  $c_2$  which is evaluated by solving both  $c_1$  and  $c_2$  concurrently. By contrast, the operator “ $\&\&$ ” denotes the usual Boolean conjunction which evaluates to either **True** or **False**.

An example of the features discussed above can be seen in the definition of a function that computes the last element of a non-empty list. The symbol “ $++$ ” denotes the usual list concatenation function:

```
last l | p++[e]=:l = e   where p,e free
```

As in Haskell, the rules defining most functions are *constructor-based* [25], in (11)  $t_1 \dots t_n$  are made of variables and/or data constructor symbols only. However, in Curry we can also use a *functional pattern* [4]. With this feature, which relies on narrowing, we can define the function **last** also as:

```
last (_++[e]) = e
```

The operational semantics of Curry, precisely described in [14,22], is a conservative extension of both lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Since Curry is based on an optimal evaluation strategy [2], it can be considered a generalization of concurrent constraint programming [27] with a lazy strategy.

Furthermore, Curry also offers features for application programming like modules, monadic I/O, encapsulated search [21], ports for distributed programming [15], libraries for GUI [16] and HTML programming [17], etc. We do not present these aspects of the language, since they are not necessary for understanding our contribution. There exist several implementations of Curry. The examples presented in this paper were all compiled and executed by PAKCS [20], a compiler/interpreter for a large subset of Curry.

There exist also other functional logic languages, most notably *TOY* [8,24], with data types, possibly non-deterministic functions, and logic variables instantiated by narrowing similar to Curry. Many patterns and exemplary programs

discussed in this paper are adaptable to these languages with minor, often purely syntactic, changes.

### 3 Patterns

In this section we present a small catalog of patterns that address non-trivial solutions of some general and challenging problems.

#### 3.1 Call-by-Reference

Name	<i>Call-by-reference</i>
Intent	return multiple values from a function without defining a containing structure
Applicability	a function must return more than one value
Structure	an argument passed to a function is an unbound variable
Consequences	avoid constructing a structure to hold multiple values
Known uses	Parser combinators
See also	Monads, Extensions

The pattern name should not mislead the reader. There is no *call-by-reference* in functional logic languages. The name stems from a similarity with the passing mode in that a value is returned by a function through an argument of the call.

When a function must return multiple values, a standard technique is to return a structure that holds all the values to be returned. For example, if function  $f$  must return both a value of type  $A$  and a value of type  $B$ , the return type could be  $(A, B)$ , a pair with components of type  $A$  and  $B$ , respectively. The client of  $f$  extracts the components of the returned structure and uses them as appropriate. Although straightforward, this approach quickly becomes tedious and produces longer and less readable code. This pattern, instead, suggests to pass unbound variables to the function which both returns a value and binds other values to the unbound variables.

**Example:** A *symbol table* is a sequence of records. A *record* is a pair in which the first component is intended as a key mapped to the second component.

```
type Record = (String,Int)
type Table = [Record]
```

The function `insert` attempts to insert a record  $(k, v)$  into a table  $t$  which is expected to contain no record with key  $k$ . This function computes both a Boolean value, `False` if a record with key  $k$  is already in  $t$ , `True` otherwise, and the updated table, if no record with key  $k$  is in  $t$ . Attempting to insert a record whose key is in the table is an error, hence the returned table in this case is uninteresting. The Boolean value is returned by the function whereas the updated table is bound to the third argument of the call. Alternatively, the function could return the updated table and bind the Boolean value to the third argument, but as we will discuss shortly this option is not as appealing.

```

insert :: Record -> Table -> Table -> Bool
insert (k,v) [] x = x := [(k,v)] &> True
insert (k,v) ((h,w):t) x
  | k == h = x := (h,w):t &> False
  | otherwise = let b = insert (k,v) t t'
                in x := (h,w):t' &> b  where t' free

```

The operator “&>”, called *constrained expression*, takes a constraint as its first argument. It solves this constraint and, if successful, returns its second argument.

The function `remove` attempts to remove a record with key *k* from a table *t* which is expected to contain one and only one such record. This function computes both a Boolean value, `True` if a record with key *k* is in *t*, `False` otherwise, and the updated table if a record with key *k* is in *t*. Attempting to remove a record whose key is not in the table is an error, hence the returned table in this case is uninteresting.

```

remove :: String -> Table -> Table -> Bool
remove _ [] = False
remove k ((h,w):t) x
  | k == h = x := t &> True
  | otherwise = let b = remove k t t'
                in x := (h,w):t' &> b  where t' free

```

An example of use of the above functions follow, where the key is a string and the value is an integer.

```

emptyTable = []
test = if insert ("x",1) emptyTable t1 &&
        insert ("y",2) t1 t2 &&
        remove "z" t2 t3 then t3
      else error "Oops"
      where t1, t2, t3 free

```

Of the two values returned by functions `insert` and `remove`, the table is subordinate to the Boolean in that when the Boolean is false, the table is not interesting. This suggest returning the Boolean from the functions and to bind the table to an argument. A client typically will test the Boolean before using the table, hence the test will trigger the binding. However, variables are bound only to fully evaluated expressions. This consideration must be taken into account to select which value to return in a variable when laziness is crucial.

For a client, it is easier to use the functions when they are coded according to the pattern rather than when they return a structure. A *state monad* [23] would be a valid alternative to this pattern for the example presented above and in other situations. Not surprisingly, this pattern can be used instead of a *Maybe* type.

This pattern is found, e.g., in the parser combinators of [7]. A parser with representation takes a sequence of tokens and typically a free variable, which is bound to the representation of the parsed tokens, whereas the parser returns the

sequence of tokens that remain to be parsed. The *Extensions* of [9] are a form of this pattern. The reference contains a comparison with the monadic approach.

This pattern is not available in functional languages since they lack free variables. Logic languages typically return information by instantiating free variables passed as arguments to predicates, but predicates do not return information, except for succeeding.

### 3.2 Many-to-Many

Name	<i>Many-to-many</i>
Intent	encode a many-to-many relation with a single simple function
Applicability	a relation is computed in both directions
Structure	a non-deterministic function defines a one-to-many relation; a functional pattern defines the inverse relation
Consequences	avoid structures to define a relation
Known uses	
See also	

We consider a many-to-many relation  $\mathcal{R}$  between two sets  $A$  and  $B$ . Some element of  $A$  is related to distinct elements of  $B$  and, vice versa, distinct elements of  $A$  are related to some element of  $B$ . In a declarative program, such a relation is typically abstracted by a function  $f$  from  $A$  to subsets of  $B$ , such that  $b \in f(a)$  iff  $a \mathcal{R} b$ . We will call this function the *core function* of the relation. Relations are dual to graphs and, accordingly, the core function can be defined, e.g., by an adjacency list. The relation  $\mathcal{R}$  implicitly defines an inverse relation which, when appropriate, is encoded in the program by a function from  $B$  to subsets of  $A$ , the core function of the inverse relation.

In this pattern, the core function is encoded as a non-deterministic function that maps every  $a \in A$  to every  $b \in B$  such that  $a \mathcal{R} b$ . The rest of the abstraction is obtained nearly automatically using standard functional logic features and libraries. In particular, the core function of the inverse relation, when needed, is automatically obtained through a functional pattern. The sets of elements related to a given element are automatically obtained using the set functions of the core function.

**Example:** Consider an abstraction about blood transfusions. We define the blood types and the function `giveTo`. The identifiers `Ap`, `An`, etc. stand for the types *A positive* ( $A+$ ), *A negative* ( $A-$ ), etc. The application `giveTo x` returns a blood type  $y$  such that  $x$  can be given to a person with type  $y$ . E.g.,  $A+$  can be given to both  $A+$  and  $AB+$ .

```
data BloodType = Ap | An | ABp | ABn | Op | On | Bp | Bn
giveTo :: BloodType -> BloodType
giveTo Ap  = Ap ? ABp
giveTo Op  = Op ? Ap ? Bp ? ABp
giveTo Bp  = Bp ? ABp
...
```

The inverse relation is trivially obtained with a function defined using a functional pattern [4]. The application `receiveFrom x` returns a blood type  $y$  such that a person with type  $x$  can receive type  $y$ . E.g.,  $AB+$  can receive  $A+$ ,  $AB+$  and  $O+$  among others.

```
receiveFrom :: BloodType -> BloodType
receiveFrom (giveTo x) = x
```

To continue the example, let us assume a database defining the blood type of a set of people, such as:

```
has :: String -> BloodType
has "John" = ABp
has "Doug" = ABn
has "Lisa" = An
```

The following function computes a donor for a patient, where the condition  $x \neq y$  avoids self-donation, which obviously is not intended.

```
donorTo :: String -> String
donorTo x
  | giveTo (has y) == has x & x /= y
  = y
  where y free
```

E.g., the application `donorTo "John"` returns both `"Doug"` and `"Lisa"`, whereas `donorTo "Lisa"` correctly fails for our very small database.

To continue the example further, we may need a particular set of donors, e.g., all the donors that live within a certain radius of a patient and we may want to rank these donors by the date of their last blood donation. For these computations, we use the set function [5] automatically defined for any function. The function `donorTo'set` produces the set of all the donors for a patient. The *SetFunctions* library module offers functions for filtering and sorting this set.

Many-to-many relations are ubiquitous, e.g., students taking courses from teachers, financial institutions owning securities, parts used to build products, etc. Often, it won't be either possible or convenient to hard-wire the relation in the program as we did in our example. In some cases, the core function of a relation will access a database or some data structure, such as a search tree, obtained from a database. An interesting application of this pattern concerns the relation among the functions of a program in which a function is related to any function that it calls. In this case, we expect that the compiler will produce a structure, e.g., a simple set of pairs, which the core function will access for its computations. Beside a small difference in the structure of the core function, the rest of the pattern is unchanged.

This pattern is not available in functional languages since they lack non-deterministic functions. Logic languages support key aspects of this pattern, in particular, the non-determinism of the core function and the possibility of computing a relation and its inverse relation with the same predicate.



### 3.3 Quantification

Name	<i>Quantification</i>
Intent	encode first-order logic formula in programs
Applicability	problems specified in a first-order logic language
Structure	apply “ <i>there exists</i> ” and “ <i>for all</i> ” library functions
Consequences	programs are encoded specifications
Known uses	
See also	

First-order logic is a common and powerful language for the specification of problems. The ability to execute even some approximation of this language enables us to directly translate many specifications into programs. A consequence of this approach is that the logic of the resulting program is correct by definition and the code is obtained with very little effort. The main hurdle is existential quantification, since specifications of this kind are often not constructive. However, narrowing, which is the most characterizing feature of functional logic languages, supports this approach.

Narrowing evaluates expressions, such as a constraint, containing free variables. The evaluation computes some instantiations of the variables that lead to the value of the expression, e.g., the satisfaction of the constraint. Hence, it solves the problem of existential quantification.

Universal quantification is more straightforward. Mapping and/or folding operations on sets are sufficient to verify whether all the elements of the set satisfy some condition. In particular, set functions can be a convenient means to compute the sets required by an abstraction.

We define the following two functions for existential and universal quantification, where `Values` is a library-defined polymorphic type abstracting a set and `mapValues` and `foldValues` are standard mapping and folding functions on sets. The function `exists` is a simple idiom defined only to improve the readability of the code.

```
exists :: a -> (a -> Success) -> Success
exists x f = f x

forall :: Values a -> (a -> Bool) -> Success
forall s f = foldValues (&&) True (mapValues f s) == True
```

**Example:** Map coloring is stated as “given any separation of a plane into contiguous regions, producing a figure called a map, ... color the regions of the map so that no two adjacent regions have the same color” [28]. A map coloring problem has a solution  $M$  iff there exists a colored map  $M$  such that for all  $x$  and  $y$  regions of  $M$  and  $x$  adjacent to  $y$  the colors of  $x$  and  $y$  differ. The above statement is a specification of the problem stated semi-formally in a first-order logic language.

We begin by defining the regions of the map and the adjacency relation. For the curious, the map is the Pacific North West.

```
data State = WA | OR | ID | BC
states = [WA,OR,ID,BC]
adjacent = [(WA,OR),(WA,ID),(WA,BC),(OR,ID),(ID,BC)]
```

To continue the example, we define the colors to use for coloring the map, only 3, and the function that colors a state. Coloring a state is a non-deterministic assignment, represented as a pair, of a color to a state.

```
data Color = Red | Green | Blue
color :: a -> (a,Color)
color x = (x, Red ? Green ? Blue)
```

The rest of the program follows:

```
solve :: [(State,Color)]
solve | exists cMap (\map ->
    forall someAdj'set (\(st1,st2) ->
        lookup st1 map /= lookup st2 map))
    = cMap
    where cMap = map color states
          someAdj = foldr1 (?) adjacent
```

The identifier `cMap` is bound to some colored map. The identifier `someAdj` is bound to some pair of adjacent states. The identifier `someAdj'set` is bound to the implicitly defined set function of `someAdj`, hence it is the set of all the pairs of adjacent states. The function `lookup` is defined in the standard *Prelude*. It retrieves the color assigned to a state in the colored map.

The condition of the function `solve` is an encoded, but verbatim, translation of the specification. The condition could be minimally shortened by eliminating the `exists` idiom, but the presented form is more readable and shows the pattern in all its generality.

Since  $\forall x P$  is equivalent to  $\neg \exists x \neg P$ , we also define:

```
notExists :: Values a -> (a -> Bool) -> Success
notExists s f = foldValues (||) False (mapValues f s) := False
```

This pattern is very general and applicable to problems, whether or not deterministic, which have non-constructive specifications. For example, the minimum element of a collection can be specified as  $m$  is the minimum of  $C$  iff there exists some  $m$  in  $C$  such that there not exists some  $x$  in  $C$  such that  $x < m$ , i.e.,  $\exists m (m \in C \wedge \neg \exists x (x \in C \wedge x < m))$  or, equivalently, for all  $x$  in  $C$ ,  $x \geq m$ .

This pattern is not available in functional languages since they lack narrowing. Logic languages have some forms of existential quantification, but their lack of functional nesting prevents the readable and elegant notion available in functional logic languages.

### 3.4 Deep Selection

Name	<i>Deep selection</i>
Intent	pattern matching at arbitrary depth in recursive types
Applicability	select an element with given properties in a structure
Structure	combine a type generator with a functional pattern
Consequences	separate structure traversal from pattern matching
Known uses	HTML and XML applications coded in Curry
See also	Curry's HTML library

Pattern matching is undoubtedly a convenient feature of modern declarative languages because it allows to easily retrieve the components of a data structure such as a tuple. Recursively defined types, such as lists and trees, have components at arbitrary depths that cannot be selected by pattern matching because pattern matching selects components only at predetermined positions. For recursively defined types, the selection of some element with a given property in a data structure typically requires code for the traversal of the structure which is intertwined with the code for using the element. The combination of functional patterns with type generators allows us to select elements arbitrarily nested in a structure in a pattern matching-like fashion without explicit traversal of the structure and mingling of different functionalities of a problem.

A list is a recursively defined type that can be used to represent a mapping by storing key-value pairs. One such structure, bound to the variable `cMap`, was used in the example of the *Quantification* pattern. The library function `lookup` retrieves from the mapping the value  $v$  associated to a key  $k$ . In that example, there is one and only one such pair in the list. The function `lookup` must both traverse the list to find the key and through pattern matching return the associated value. The two computations are intermixed and pattern matching some element with different characteristic in a list would require duplication of the code to traverse the list. Functional patterns offer a new option.

The key idea of the *deep selection* pattern is to define a “generator” function that generates all the instances of a type with a given element. This function is interesting for recursively defined types. For a list, this generator is:

```
withElem :: a -> [a]
withElem e = e:unknown ? unknown:withElem e
```

The function `unknown` is defined in the *Prelude* and simply returns a free variable. This generator supports the following definition of `lookup` in which the (functional) pattern is as simple as it can be.

```
lookup :: :: [(a,b)] -> b
lookup (withElem (_,v)) = v
```

The counterpart of `withElem` is `elemOf`, an “extractor” as opposed to a generator, which returns non-deterministically a component of a structure:

```
elemOf :: [a] -> a
elemOf (withElem e) = e
```

We will use both these functions including specialized variations of them.

**Example:** Below, we show a simple type for representing arithmetic expressions and a generator of all the expressions with a given subexpression:

```
data Exp = Lit Int
        | Var [Char]
        | Add Exp Exp
        | Mul Exp Exp

withSub :: Exp -> Exp
withSub exp = exp
            ? op (withSub exp) unknown
            ? op unknown (withSub exp)
  where op = Add ? Mul
```

Suppose that we want to find all the variables of an expression. The function `varOf`, a specialization of `elemOf` shown earlier, for the type `Exp`, takes an expression `exp` and returns the identifier of some variable of `exp`.

```
varOf :: Exp -> String
varOf (withSub (Var v)) = v
```

The set of identifiers of all the variables of `exp` is simply obtained with the set function of `varOf`, i.e., `varOf'set exp`.

In some situations, a bit more machinery is needed. For example, suppose that we want to find common subexpressions of an expression, such as 42 and `y` in the following:

```
Add (Mul (Lit 42) (Add (Lit 42) (Var "y")))
    (Add (Var "x") (Var "y"))
```

One option is a more specialized generator that generates all and only the expressions with a given common subexpression:

```
withCommonSub :: Exp -> Exp
withCommonSub exp = op (withCommonSub exp) unknown
                  ? op unknown (withCommonSub exp)
                  ? op (withSub exp) (withSub exp)
  where op = Add ? Mul
```

Another option is a different more specialized generator that generates all the expressions with a given subexpression at a given position. The position is a string of 1's and 2's defining a path from the root of the expression to the subexpression.

```
withSubAt :: [Int] -> Exp -> Exp
withSubAt [] exp = exp
withSubAt (1:ps) exp = (Add ? Mul) (withSubAt ps exp) unknown
withSubAt (2:ps) exp = (Add ? Mul) unknown (withSubAt ps exp)
```

This generator is useful to pattern match a subexpression and its position:

```
subAt :: Exp -> ([Int],Exp)
subAt (withSubAt p exp) = (p,exp)
```

In the new version of the function that computes a common subexpression, not only we return the common subexpression, but also the two positions at which subexpression occurs, since they are available. The ordering operator “<:” is predefined for all types. Its use in our code ensures that the same subexpression is not matched twice.

```
commonSub :: Exp -> (Exp,[Int],[Int])
commonSub exp | p1 <: p2 & e1:=e2 = (e1,p1,p2)
      where (p1,e1) = subAt exp
            (p2,e2) = subAt exp
```

This pattern is applied in HTML processing. Curry provides a library for the high-level construction of type-safe HTML documents and web-oriented user interfaces [18]. HTML documents are instances of a type `HtmlExp`, shown below, consisting of sequences of text and tag elements with both attributes and possibly nested elements.

```
data HtmlExp = HtmlText String
              | HtmlStruct String [(String,String)] [HtmlExp]
```

The problem, sought-after by spammers, of finding all the e-mail addresses in a HTML page is trivialized by this pattern. The following function finds some e-mail address in a document:

```
eAddress :: HtmlExp -> String
eAddress (withHtmlElem
          (HtmlStruct _
            (withElem ("href","mailto:++name")) _)) = name
```

where `withElem`, defined above, is applied to match a `href` tag with value `mailto` in a list of attributes and the type generator `withHtmlElem`, defined below, is applied to match an `HtmlStruct` structure with the desired attribute in a tree of HTML structures.

```
withHtmlElem :: HtmlExp -> HtmlExp
withHtmlElem helem = helem
                  ? HtmlStruct unknown
                    unknown
                    (withElem (withHtmlElem helem))
```

All the addresses in a page are produced by the set function of `eAddress`.

In a similar way, one can also define generators for deep matching in XML structures. A library to support the declarative processing of XML data based on the deep selection pattern is described in [19].

This pattern is not available in both functional and logic languages since they lack functional patterns.

### 3.5 Non-determinism Introduction and Elimination

Name	<i>Non-determinism introduction and elimination</i>
Intent	use different algorithms for the same problem
Applicability	some algorithm is too slow or it may be incorrect
Structure	either replace non-deterministic code with deterministic one or vice versa
Consequences	improve speed or verify correctness of algorithms
Known uses	prototyping
See also	

Specifications of problems are often non-deterministic because in many cases non-determinism defines the desired results of a computation more easily than by other means. We have seen this practice in several previous examples. Functional logic programming, more than any other paradigm, allows a programmer to translate a specification, whether or not non-deterministic, with little or no change into a program [1]. Thus, it is not unusual for programmers to initially code non-deterministic programs even for deterministic problems because this approach produces correct programs quickly. We call a *prototypical implementation* this direct encoding of a specification.

For some problems, prototypical implementations are not as efficient as an application requires. This is typical, e.g., for sorting and searching problems, which have been the subject of long investigations, because non-deterministic solutions ignore domain knowledge that speeds up computations. In these cases, the prototypical implementation, often non-deterministic, should be replaced by a more efficient implementation, often deterministic, that produces the same result. We call the latter *production implementation*. The investment that went into the prototypical implementation is not wasted, as several benefits derive from that effort. First of all, the specification of the problem is better understood and it has been tested through the input/output behavior of the prototypical implementation and possibly debugged and corrected. Second, the prototypical implementation can be used as a testing oracle of the production implementation. Testing can be largely automated, which both reduces effort and improves reliability.

PAKCS [20] is distributed with a unit testing tool [13] called *CurryTest* which is useful in the situation we describe. A unit testing of a program is another program defining, among others, some zero-arity functions containing assertions, e.g., specific conditions stating the equality between a function call and its result. The *CurryTest* tool applied to the program invokes, using reflections, all the functions defining an assertion. The tool checks the validity of each assertion and reports any violation. Thus, a test of a function  $f$  of the production implementation will apply both  $f$  and the corresponding function of the prototypical

implementation to test arguments, and assert that the results are the same.

**Example:** Assume an informal specification of sorting: “find the minimum of a list, sort the rest, and place the minimum at the front.” The most complicated aspect of an implementation of this specification is the computation of the minimum. A fairly precise specification of the minimum was given in the *Quantification* pattern. A prototypical implementation of this specification follows:

```
getMinSpec :: [a] -> a
getMinSpec l | exists m (\m -> m == elemOf l &>
                        notExists (elemOf'set l) (\x -> x < m))
              = m  where m free
```

The prototypical implementation is assumed to be correct, since it is a direct encoding of the specification, and it is non-deterministic due to `elemOf`. Sorting with the prototypical implementation is too slow for long lists, hence we code a deterministic and more efficient production implementation:

```
getMin :: [a] -> a
getMin (x:xs) = aux x xs
               where aux x [] = x
                     aux x (y:ys) | x <= y = aux x ys
                                   | otherwise = getMin (y:ys)
```

To test the production implementation, we define the following function that compares the output of the production implementation with that of the prototypical implementation. We process this function with *CurryTest*.

```
testGetMin :: Assertion Int
testGetMin = AssertEqual "getMin" outSpec out
               where input = [3,1,2,4,9,5] -- some test data
                     outSpec = getMinSpec input
                     out      = getMin      input
```

This pattern is typically used to replace non-deterministic code with more deterministic code to improve the efficiency of a program. However, the opposite replacement is occasionally useful to attempt to “improve the correctness” of a program. Suppose that the input/output behavior of a program is incorrect and that we suspect that the culprit is some function  $f$ . We can replace the code of  $f$  with code directly obtained from the specification of  $f$ . This code is likely to be more non-deterministic. If the replacement fixes the input/output behavior of the program, we have the proof that the code that we replaced was indeed incorrect.

Both unit testing and replacing an algorithm with another for various purposes, such as improving performance or verifying behavior, are widespread and language independent techniques. The customization of these techniques to functional logic programming emphasizes the possibility of executing non-deterministic code, in particular code obtained from a direct encoding of a specification.

## 4 Conclusion and Related Work

Design patterns help structuring code for general problems frequently arising in software development. They produce solutions that are more readable, maintainable and elegant than improvised alternatives. Efficiency may be partially sacrificed for these very desirable attributes. We have also shown that employing a pattern has various benefits even in situations in which the pattern-driven solution is inefficient and it is eventually replaced by more efficient code.

We presented five new design patterns for the functional logic programming paradigm. These patterns are a follow up on our initial work [3] in this area. Patterns distill successful programming experience similar in scope to programming pearls. Some of our patterns were motivated, in part, by features introduced in the Curry language in the last 10 years, in particular functional patterns [4] and set functions [5].

The programs discussed in this paper are available at URL:

<http://www.cs.pdx.edu/~antoy/flp/patterns/>

## References

1. Antoy, S.: Programming with narrowing. *Journal of Symbolic Computation* 45(5), 501–522 (2010)
2. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. *Journal of the ACM* 47(4), 776–822 (2000)
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. Antoy, S., Hanus, M.: Declarative programming with function patterns. In: Hill, P.M. (ed.) *LOPSTR 2005*. LNCS, vol. 3901, pp. 6–22. Springer, Heidelberg (2006)
5. Antoy, S., Hanus, M.: Set functions for functional logic programming. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, Lisbon, Portugal, pp. 73–82 (September 2009)
6. Beck, K., Cunningham, W.: Using pattern languages for object-oriented programs. In: *Specification and Design for Object-Oriented Programming, OOPSLA 1987* (1987)
7. Caballero, R., López-Fraguas, F.: A functional-logic perspective of parsing. In: Mideldorp, A. (ed.) *FLOPS 1999*. LNCS, vol. 1722, pp. 85–99. Springer, Heidelberg (1999)
8. Caballero, R., Sánchez, J. (eds.): *TOY: A Multiparadigm Declarative Language, version 2.3.1* (2007), <http://toy.sourceforge.net>
9. Caballero, R., López-Fraguas, F.J.: Extensions: A technique for structuring functional-logic programs. In: Björner, D., Broy, M., Zamulin, A.V. (eds.) *PSI 1999*. LNCS, vol. 1755, pp. 297–310. Springer, Heidelberg (2000)
10. Cooper, J.W.: *Java Design Patterns*. AddisonWesley, London (2000)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesley, London (1994)
12. Grand, M.: *Patterns in Java*. J. Wiley, Chichester (1998)



13. Hanus, M.: Currytest: A tool for testing Curry programs,  
<http://www-ps.informatik.uni-kiel.de/currywiki/tools/currytest>  
 (accessed April 13, 2011)
14. Hanus, M.: A unified computation model for functional and logic programming. In: Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris), pp. 80–93 (1997)
15. Hanus, M.: Distributed programming in a multi-paradigm declarative language. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 376–395. Springer, Heidelberg (1999)
16. 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)
17. Hanus, M.: High-level server side web scripting in curry. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 76–92. Springer, Heidelberg (2001)
18. 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)
19. Hanus, M.: Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel (2011)
20. 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 (2011),  
<http://www.informatik.uni-kiel.de/~pakcs/>
21. Hanus, M., Steiner, F.: Controlling search in declarative programs. In: Palamidessi, C., Meinke, K., Glaser, H. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 374–390. Springer, Heidelberg (1998)
22. Hanus, M. (ed.): Curry: An integrated functional logic language (vers. 0.8.2) (March 28, 2006), <http://www.informatik.uni-kiel.de/~curry>
23. Hudak, P., Peterson, J., Fasel, J.: A gentle introduction to Haskell 98 (1999),  
<http://www.haskell.org/tutorial/monads.html>
24. Fraguas, F.J.L., Hernández, J.S.: TOY: A multiparadigm declarative system. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
25. O’Donnell, M.J.: Equational Logic as a Programming Language. MIT Press, Cambridge (1985)
26. Peyton Jones, S.L., Hughes, J.: Haskell 98: A non-strict, purely functional language (1999), <http://www.haskell.org>
27. Saraswat, V.A.: Concurrent Constraint Programming. MIT Press, Cambridge (1993)
28. Wikipedia, the free encyclopedia. Four color theorem,  
[http://en.wikipedia.org/wiki/Four\\_color\\_theorem](http://en.wikipedia.org/wiki/Four_color_theorem) (accessed April 8, 2011)

# XQuery in the Functional-Logic Language Toy

Jesus M. Almendros-Jiménez<sup>1</sup>, Rafael Caballero<sup>2</sup>, Yolanda García-Ruiz<sup>2</sup>,  
and Fernando Sáenz-Pérez<sup>3,\*</sup>

<sup>1</sup> Dpto. de Lenguajes y Computación, Universidad de Almería

<sup>2</sup> Departamento de Sistemas Informáticos y Computación  
Universidad Complutense de Madrid

<sup>3</sup> Departamento de Ingeniería del Software e Inteligencia Artificial  
Universidad Complutense de Madrid  
Spain

**Abstract.** This paper presents an encoding of the XML query language XQuery in the functional-logic language  $\mathcal{TOY}$ . The encoding is based on the definition of for-let-where-return constructors by means of  $\mathcal{TOY}$  functions, and uses the recently proposed XPath implementation for this language as a basis. XQuery expressions can be executed in  $\mathcal{TOY}$  obtaining sequences of XML elements as answers. Our setting exploits the non-deterministic nature of  $\mathcal{TOY}$  by retrieving the elements of the XML tree once at a time when necessary. We show that one of the advantages of using a rewriting-based language for implementing XQuery is that it can be used for optimizing XQuery expressions by query rewriting. With this aim, XQuery expressions are converted into higher order patterns that can be analyzed and modified by  $\mathcal{TOY}$  functions.

**Keywords:** Functional-Logic Programming, Non-Deterministic Functions, XQuery, Higher-Order Patterns.

## 1 Introduction

In the last few years the eXtensible Markup Language XML [33] has become a standard for the exchange of semistructured data. Thus, querying XML documents from different languages has become a convenient feature. XQuery [35,37] has been defined as a query language for finding and extracting information from XML documents. It extends XPath [34], a domain-specific language that has become part of general-purpose languages. Recently, in [10], we have proposed an implementation of XPath in the functional-logic language  $\mathcal{TOY}$  [22]. The implementation is based on the definition of XPath constructors by means of  $\mathcal{TOY}$  functions. As well, XML documents are represented in  $\mathcal{TOY}$  by means of terms, and the basic constructors of XPath: `child`, `self`, `descendant`, etc.

---

\* This work has been supported by the Spanish projects TIN2008-06622-C03-01, TIN2008-06622-C03-03, S-0505/TIC/0407, S2009TIC-1465, and UCM-BSCH-GR58/08-910502.

are defined as functions that apply to XML terms. The goal of this paper is to extend [10] to XQuery.

The existing XQuery implementations either use functional programming or Relational Database Management Systems (RDBMS's). In the first case, the *Galax* implementation [23] encodes XQuery into *Objective Caml*, in particular, encodes XPath. Since XQuery is a functional language (with some extensions) the main encoding is related with the type system for allowing XML documents and XPath expressions to occur in a functional expression. With this aim, a specific type system for handling XML tags, the hierarchical structure of XML, and sequences of XML items is required. In addition, XPath expressions can be implemented from this representation. There are also proposals for new languages based on functional programming rather than implementing XPath and XQuery. This is the case of *XDuce* [19] and *CDuce* [5,6], which are languages for XML data processing, using regular expression pattern matching over XML trees and subtyping as basic mechanism. There are also proposals around *Haskell* for handling XML documents, such as *HaXML* and *UUXML* [31,4,36,30]. XML types are encoded with Haskell's type classes providing a Haskell library in which XML types are encoded as algebraic datatypes. *HXQ* [14] is a translator from XQuery to embedded Haskell code, using the Haskell templates. *HXQ* stores XML documents in a relational database, and translates queries into SQL queries.

This is also followed in some RDBMS XQuery implementations: XML documents are encoded with relational tables, and XPath and XQuery with SQL. The most relevant contribution in this research line is *MonetDB/XQuery* [7]. It consists of the *Pathfinder* XQuery compiler [8] on top of the *MonetDB* RDBMS, although *Pathfinder* can be deployed on top of any RDBMS. *MonetDB/XQuery* encodes the XML tree structure in a relational table following a pre/post order traversal of the tree (with some variant). XPath can be implemented from such table-based representation, and XQuery by encoding *flwor* expressions into the *relational algebra*, extended with the so-called *loop-lifted staircase join*.

There are also proposals based on logic programming. In most cases, new languages for XML processing are proposed. The *Xcerpt* project [27,9] 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. Another contribution to XML processing is the language *XPathLog* (integrated in the the *Lopix* system) [24] which is a *Datalog*-style extension for *XPath* with variable bindings. *XCentric* [13] is an approach for representing and handling XML documents by logic programs, by considering terms with functions of flexible arity and regular types. *XPath<sup>L</sup>* [26] is a logic language based on rules for XML processing including a specific predicate for handling *XPath* expressions in *Datalog* programs. *FNPath* [29] 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 either consist of facts (graph notation) or a term structure (field notation). *FNPath* can evaluate XPath expressions based on that DOM. [2,3] aim to implement XQuery by means of logic programming, providing two

alternatives: a top-down and a bottom-up approaches (the latter in the line of Datalog programs). Finally, some well-known *Prolog* implementations include libraries for loading XML documents, such as *SWI-Prolog* [38] and *Ciao* [12].

In the field of functional-logic languages, [18] proposes a rule-based language for processing semistructured data that is implemented and embedded in the functional logic language Curry [17]. The framework is based on providing operations to describe partial matchings in the data and exploits functional patterns and set functions for the programming tasks.

In functional and functional-logic languages, a different approach is possible: XPath queries can be represented by higher-order functions connected by higher-order combinators. Using this approach, an XPath query becomes at the same time implementation (code) and representation (data term). This is the approach we have followed in our previous work [10]. In the case of XQuery, forget-where-return constructors can be encoded in  $\mathcal{TOY}$ , which uses the XPath query language as a basis. XQuery expressions can be encoded by means of (first-order) functions. However, we show that we can also consider XQuery expressions as higher order patterns, in order to manipulate XQuery programs by means of  $\mathcal{TOY}$ . For instance, we have studied how to transform XQuery expressions into  $\mathcal{TOY}$  patterns in order to optimize them. In this paper we follow this idea, which has been used in the past, for instance for defining parsers in functional and functional-logic languages [11,20]. A completely declarative proposal for integrating part of XQuery in  $\mathcal{TOY}$  can be found in [1], which restricts itself to the completely declarative features of the language. This implies that the subset of XQuery considered is much narrower than the framework presented here. The advantage of restricting to the purely declarative view is that proofs of correctness and completeness are provided. In this work we take a different point of view, trying to define a more general XQuery framework although using non-purely declarative features as the (meta-)primitive `collect`. Another difference of this work is the use of higher-order patterns for rewriting queries, which was not available in [1].

The specific characteristics of functional-logic languages match perfectly the nature of XQuery queries:

- *Non-deterministic functions* are used to nicely represent the evaluation of an XPath/XQuery query, which consists of fragments of the input XML document. In addition, the `for` constructor of XQuery can be defined with non-deterministic behavior.
- *Logic variables* are employed for instance when obtaining the contents of XPath text nodes, and for solving nested XQuery expressions, capturing the non-deterministic behavior of inner `for` and XPath expressions.
- By defining rules with *higher-order patterns*, XPath/XQuery queries become truly first-class citizens in our setting. In the case of XQuery, this allows us to rewrite queries in order to be optimized. XPath can also be optimized (see [10] for more details).

The rest of the paper is organized as follows. Section 2 briefly introduces the XPath subset presented in [10]. Section 3 defines the encoding of XQuery in

$\mathcal{TOY}$ . Section 4 shows how to use  $\mathcal{TOY}$  for the optimization of XQuery. Finally, Section 5 presents some conclusions.

## 2 XPath in $\mathcal{TOY}$

This section introduces the functional-logic language  $\mathcal{TOY}$  [22] and the subset of XPath that we intend to integrate with  $\mathcal{TOY}$ , omitting all the features of XPath that are supported by  $\mathcal{TOY}$  but not used in this paper, such as filters, abbreviations, attributes and preprocessing of reverse axes. See [10] for a more detailed introduction to XPath in  $\mathcal{TOY}$ .

### 2.1 The Functional-Logic Language $\mathcal{TOY}$

All the examples in this paper are written in the concrete syntax of the lazy functional-logic language  $\mathcal{TOY}$  [22], but most of the code can be easily adapted to other similar languages as Curry [17].  $\mathcal{TOY}$  is a lazy functional-logic language. A  $\mathcal{TOY}$  program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax is similar to the functional language Haskell, except for the capitalization, which follows the approach of Prolog (variables start by uppercase, and other symbols by lowercase [1]). Each rule for a function  $f$  has the form:

$$\underbrace{f \ t_1 \dots t_n}_{\text{left-hand side}} = \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{e_1, \dots, e_k}_{\text{condition}} \text{ where } \underbrace{s_1 = u_1, \dots, s_m = u_m}_{\text{local definitions}}$$

where  $u_i$  and  $r$  are expressions (that can contain new extra variables) and  $t_i$ ,  $s_i$  are patterns. The overall idea is that a function call ( $f \ e_1 \dots e_n$ ) returns an instance  $r\theta$  of  $r$ , if:

- Each  $e_i$  can be reduced to some pattern  $a_i$ ,  $i = 1 \dots n$ , such that ( $f \ t_1 \dots t_n$ ) and ( $f \ a_1 \dots a_n$ ) are unifiable with most general unifier  $\theta$ , and
- $u_i\theta$  can be reduced to pattern  $s_i\theta$  for each  $i = 1 \dots m$ .

Infix operators are also allowed as particular case of program functions. Consider for instance the definitions:

<code>infixr 30 /\</code> <code>false /\ X = false</code> <code>true /\ X = X</code>	<code>infixr 30 \/</code> <code>true \/ X = true</code> <code>false \/ X = X</code>	<code>infixr 45 ?</code> <code>X ? _Y = X</code> <code>_X ? Y = Y</code>
--	---	--

The  $\wedge$  and  $\vee$  operators represent the standard conjunction and disjunction, respectively, while  $?$  represents the non-deterministic choice. For instance the infix declaration `infixr 45 ?` indicates that  $?$  is an infix operator that associates to the right (the  $r$  in `infixr`) and that its priority is 35. The priority is used to assume precedences in the case of expressions involving different operators. Computations in  $\mathcal{TOY}$  start when the user inputs some goal as

<sup>1</sup> Also, only variables are allowed to start that way. If another identifier has to start with uppercase or underscore, it must be delimited between single quotes.

```
Toy> 1 ? 2 ? 3 ? 4 == R
```

This goal asks  $\mathcal{TOY}$  for values of the logical variable  $R$  that make true the (strict) equality  $1 ? 2 ? 3 ? 4 == R$ . This goal yields four different answers  $\{R \mapsto 1\}$ ,  $\{R \mapsto 2\}$ ,  $\{R \mapsto 3\}$ , and  $\{R \mapsto 4\}$ . The next function extends the choice operator to lists: `member [X|Xs] = X ? member Xs`. For instance, the goal `member [1,2,3,4] == R` has the same four answers that were obtained by trying `1 ? 2 ? 3 ? 4 == R`.

$\mathcal{TOY}$  is a *typed* language. Types do not need to be annotated explicitly by the user, they are inferred by the system, which rejects ill-typed expressions. However, function type declarations can also be made explicit by the user, which improves the clarity of the program and helps to detect some bugs at compile time. For instance, a function type declaration is: `member :: [A] -> A` which indicates that `member` takes a list of elements of type  $A$ , and returns a value which must be also of type  $A$ . As usual in functional programming languages,  $\mathcal{TOY}$  allows partial applications in expressions and higher order parameters like `apply F X = F X`. Consider for instance the function that returns the  $n$ -th value in a list:

```
nth :: int -> [A] -> A
nth N [X|Xs] = if N==1 then X else nth (N-1) Xs
```

This function has program arity 2, which means that the program rule is applied when it receives `nth 1 == R1`, `R1 ["hello","friends"] == R2` and produces the answer  $\{R1 \mapsto (\text{nth } 1), R2 \mapsto \text{"hello"}\}$ . In this solution,  $R1$  is bound to the partial application `nth 1`. Observe that  $R1$  has type  $([A] \rightarrow A)$ , and thus it is a *higher-order* variable. Applying  $R1$  to a list of strings like in the second part of the goal `R1 ["hello","friends"] == R2` 'triggers' the use of the program rule for `nth`. A particularity of  $\mathcal{TOY}$  is that partial applications with pattern parameters are also valid patterns. They are called *higher-order patterns*. For instance, a program rule like `foo (apply member) = true` is valid, although `foo (apply member []) = true` is not because `apply member []` is a reducible expression and not a valid pattern. For instance, one could define a function like: `first (nth N) = N==1` because `nth N` is a higher-order pattern. However, a program rule like: `foo (nth 1 [2]) = true` is not valid, because `(nth 1 [2])` is reducible and thus it is not a valid pattern. Higher-order variables and patterns play an important role in our setting.

## 2.2 Representing XPath Queries

Data type declarations and type alias are useful for representing XML documents in  $\mathcal{TOY}$ , as illustrated next:

```
data xmlNode      = txt      string
                  | comment string
                  | xmlTag   string [xmlAttribute] [xmlNode]
data xmlAttribute = att      string string
```

```

type xml          = xmlNode
type xPath        = xml -> xml

```

Data type `xmlNode` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, comments, and tags (element nodes), each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, constructor `xmlTag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. Data type `xmlAttribute` contains the name of the attribute and its value (both of type `string`). Type alias `xml` is a renaming of the data type `xmlNode`. Finally, type alias `xPath` is defined as a function from nodes to nodes, and is the type of XPath constructors. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation. Notice that in *TOY* we do not still consider the adequacy of the document to its underlying *Schema* definition [32]. This task has been addressed in functional programming defining regular expression types [30]. However, we assume well-formed input XML documents. In order to import XML documents, the *TOY* primitive `load_xml_file` loads an XML file returning its representation as a value of type `xmlNode`. Figure 1 shows an example of XML file and its representation in *TOY*.

Typically, XPath expressions return several fragments of the XML document. Thus, the expected type in *TOY* for `xPath` could be `type xPath = xml -> [xml]` meaning that a list or sequence of results is obtained. This is the approach considered in [2] and also the usual in functional programming [16]. However,

<pre> &lt;?xml version='1.0'?&gt; &lt;food&gt;   &lt;item type="fruit"&gt;     &lt;name&gt;watermelon&lt;/name&gt;     &lt;price&gt;32&lt;/price&gt;   &lt;/item&gt;   &lt;item type="fruit"&gt;     &lt;name&gt;oranges&lt;/name&gt;     &lt;variety&gt;navel&lt;/variety&gt;     &lt;price&gt;74&lt;/price&gt;   &lt;/item&gt;   &lt;item type="vegetable"&gt;     &lt;name&gt;onions&lt;/name&gt;     &lt;price&gt;55&lt;/price&gt;   &lt;/item&gt;   &lt;item type="fruit"&gt;     &lt;name&gt;strawberries&lt;/name&gt;     &lt;variety&gt;alpine&lt;/variety&gt;     &lt;price&gt;210&lt;/price&gt;   &lt;/item&gt; &lt;/food&gt; </pre>	<pre> xmlTag "root" [att "version" "1.0"] [ xmlTag "food" [] [   xmlTag "item" [att "type" "fruit"] [     xmlTag "name" [] [txt "watermelon"],     xmlTag "price" [] [txt "32"]   ],   xmlTag "item" [att "type" "fruit"] [     xmlTag "name" [] [txt "oranges"],     xmlTag "variety" [] [txt "navel"],     xmlTag "price" [] [txt "74"]   ],   xmlTag "item" [att "type" "vegetable"] [     xmlTag "name" [] [txt "onions"],     xmlTag "price" [] [txt "55"]   ],   xmlTag "item" [att "type" "fruit"] [     xmlTag "name" [] [txt "strawberries"],     xmlTag "variety" [] [txt "alpine"],     xmlTag "price" [] [txt "210"]   ] ]] </pre>
--	--

Fig. 1. XML example (left) and its representation in *TOY* (right)

in our case we take advantage of the non-deterministic nature of our language, returning each result individually. We define an XPath expression as a function taking a (fragment of) XML as input and returning a (fragment of) XML as its result: `type xPath = xml -> xml`. In order to apply an XPath expression to a particular document, we use the following infix operator definition:

```
(<--> :: string -> xPath -> xml      S <-- Q = Q (load_xml_file S)
```

The input arguments of this operator are a string `S` representing the file name and an XPath query `Q`. The function applies `Q` to the XML document contained in file `S`. This operator plays in  $\mathcal{TOY}$  the role of `doc` in XPath. The XPath combinators `/` and `::` which correspond to the connection between steps and between axis and tests, respectively, are defined in  $\mathcal{TOY}$  as function composition:

```
infixr 55 ::..                               infixr 40 ./.  
(::.. :: xPath -> xPath -> xPath (./.) :: xPath -> xPath -> xPath  
(F ::.. G) X = G (F X)                      (F ./.. G) X = G (F X)
```

We use the function operator names `::..` and `./..` because `::` and `/` are already defined in  $\mathcal{TOY}$ . Also notice that their definitions are the same. Indeed, we could use a single operator for representing both combinators, but we decided to do this way for maintaining a similar syntax for XPath practitioners, more accustomed to use such symbols. In addition, we do not check for the “appropriate” use of such operators and either rely on the provided automatic translation by the parser or left to the user. The variable `X` represents the input XML fragment (the context node). The rules specify how the combinator applies the first XPath expression (`F`) followed by the second one (`G`). Figure 2 shows the  $\mathcal{TOY}$  definition of XPath main axes and tests. `node`. In our setting, it corresponds simply to the identity function. A more interesting axis is `child`, which returns, using the non-deterministic function `member`, all the children of the context node. Observe that in XML only *element nodes* have children, and that in the  $\mathcal{TOY}$  representation these nodes correspond to terms rooted by constructor `xmlTag`. Once `child` has been defined, `descendant` and `descendant-or-self` are just generalizations. The first rule for this function specifies that `child` must be used once, while the second rule corresponds to two or more applications of `child`. In this rule,

<pre>self,child,descendant :: xPath descendant_or_self :: xPath self X = X child (tag _ _ L) = member L descendant X = child X descendant X = if child X == Y                 then descendant Y descendant_or_self =                 self ? descendant</pre>	<pre>nodeT,elem :: xPath nameT,textT,commentT::string-&gt;xPath nodeT X = X nameT S (xmlTag S Att L) =                 xmlTag S Att L textT S (txt S) = txt S commentT S (comment S) = comment S elem = nameT _</pre>
--	---

Fig. 2. XPath axes and tests in  $\mathcal{TOY}$



the `if` statement is employed to ensure that `child` succeeds when applied to the input XML fragment, thus avoiding possibly infinite recursive calls. Finally, the definition of axis `descendant-or-self` is straightforward. Observe that the XML input argument is not necessary in this natural definition. With respect to test nodes, the first test defined in Figure 2 is `nodeT`, which corresponds to `node()` in the usual XPath syntax. This test is simply the identity. For instance, here is the XPath expression that returns all the nodes in an XML document, together with its *TOY* equivalent:

```
XPath → doc("food.xml")/descendant-or-self::node()
TOY   → ("food.xml" <- descendant_or_self::..nodeT)==R
```

The only difference is that the *TOY* expression returns one result at a time in the variable `R`, asking the user if more results are needed. If the user wishes to obtain all the solutions at a time, as usual in XPath evaluators, then it is enough to use the primitive `collect`. For instance, the answer to the *TOY* goal:

```
Toy> collect ("food.xml" <-- descendant_or_self::..nodeT) == R
```

produces a single answer, with `R` instantiated to a list whose elements are the nodes in "food.xml". XPath abbreviated syntax allows the programmer to omit the axis `child::` from a location step when it is followed by a name. Thus, the query `child::food/child::price/child::item` simply `food/price/item`. In *TOY* we cannot do that directly because we are in a typed language and the combinator `./.` expects xPath expressions and not strings. However, we can introduce a similar abbreviation by defining new unitary operators `name` (and similarly `text`), which transform strings into XPath expressions:

```
name :: string -> xPath
name S = child::..(nameT S)
```

So, we can write in *TOY* `name "food" ./ .name "item" ./ .name "price"`.

Other tests as `nameT` and `textT` select fragments of the XML input, which can be returned in a logical variable, as in:

```
XPath → child::food/child::item/child::price/child::text()
TOY   → child::..nameT "food" ./ .child::..nameT "item" ./ .
        child::..nameT "price" ./ .child::..textT P
```

The logic variable `P` obtains the prices contained in the example document. Another XPath useful abbreviation is `//` which stands for the unabbreviated expression `/descendant-or-self::node()/`. In *TOY*, we can define:

```
infixr 30 ./ .
(./.) :: xPath -> xPath -> xPath
A ./ . B = append A (descendant_or_self ::.. nodeT ./ . B)
append :: xPath -> xPath -> xPath
append (A::..B) C = (A::..B) ./ . C
append (X ./ .Y) C = X ./ . (append Y C)
```

Notice that a new function **append** is used for concatenating XPath expressions. This function is analogous to the well-known **append** for lists, but defined over **xPath** terms. This is our first example of the usefulness of higher-order patterns since for instance pattern  $(A :: .B)$  has type **xPath**, i.e.,  $\text{xml} \rightarrow \text{xml}$ .

### 3 XQuery in $\mathcal{TOY}$

Now, we are in a position to define the proposed extension to XQuery. Firstly, the subset of XQuery expressions handled in our setting is presented (XQuery is a richer language than the fragment presented here):

```
XQuery ::= XPath | $Var | XQuery/XPath* |
        let $Var := XQuery [where BXQuery] return XQuery |
        for $Var in XQuery [where BXQuery] return XQuery |
        <tag> XQuery </tag>
```

```
BXQuery ::= XQuery | XQuery=XQuery
```

Basically, the XQuery fragment handled in  $\mathcal{TOY}$  allows building new XML documents employing new tags, and the traversal of XML documents by means of the **for** construction. XQuery variables are used in **for** and **let** expressions and can occur in the built documents and XPath expressions. It is worth observing that XPath can be applied to XQuery expressions, that is, for instance, XPath can be applied to the result of a **for** expression. Therefore, such XPath expressions are not rooted by documents (they are denoted by XPath\*). In order to encode XQuery in  $\mathcal{TOY}$  we define a new type:

```
type xQuery = [xml]
```

In Section 2, XPath has been represented as functions from xml nodes to xml nodes. However, XQuery expressions are defined as sequences of xml nodes represented in  $\mathcal{TOY}$  by lists. This does not imply that our approach returns the answers enclosed in lists, it still uses non-determinism for evaluating **for** and XPath expressions. We define functions for representing **for-let-where-return** expressions as follows. Firstly, **let** and **for** expressions are defined as:

```
xLet :: xQuery -> xQuery -> xQuery
xLet X [Y]           = if X == collect Y then X
xLet X (X1:X2:L) = if X == (X1:X2:L) then X
```

```
xFor :: xQuery -> xQuery -> xQuery
xFor X [Y]           = if X == [Y] then X
xFor X (X1:X2:L) = if X == [member (X1:X2:L)] then X
```

**xLet** uses **collect** for capturing the elements of **Y** in a list, whereas **xFor** retrieves non deterministically the elements of **Expr** in unitary lists. It fits well, for instance, when **Y** is an XPath expression in  $\mathcal{TOY}$ . The definition of **for** relies on the non-deterministic function **member** defined in Section 2. Now  $\mathcal{TOY}$  goals like **xFor X ("food.xml" <\$- name "food" ./ . name "item")=R** or **xLet X**

("food.xml" <\$- name "food" ./ . name "item")==R can be tried. Let us remark that XPath expressions have been modified in XQuery as follows. A new operator <\$- is defined in terms of <-:

```
infixr 35 <$--
(<$--) :: string -> XPath -> XQuery
(<$--) Doc Path = [(<--) Doc Path]
```

The function <\$- returns (non deterministically) unitary lists with the elements of the given document in the corresponding path. Therefore, XPath and for expressions have the same behavior in the  $\mathcal{TOY}$  implementation of XQuery. In other words, (<\$-) serves for type conversion from XPath to XQuery. Now, we can define **where** and **return** as follows:

```
infixr 35 'xWhere'
('xWhere') :: XQuery -> bool -> XQuery
('xWhere') X Y = if Y then X

infixr 35 'xReturn'
('xReturn') :: XQuery -> XQuery -> XQuery
('xReturn') X Y = if X == _ then Y
```

The definition of **xWhere** is straightforward: the query **X** is returned if the condition **Y** can be satisfied. The **if** statement in **xReturn** forces the evaluation of **X**. The anonymous variable (**\_**) can be read as *if the query X does not fail, then return Y*. With these definitions, we can simulate many XQuery expressions in  $\mathcal{TOY}$ . However, there are two elements still to be added. XPath expressions can now be rooted by XQuery expressions. Thus, we add a new function:

```
infixr 35 <$
(<$) :: XQuery -> XPath -> XQuery
(<$) [Y] Path = [Path Y]
(<$) (X:Y:L) Path = map Path (X:Y:L)
```

The first argument is an XPath variable or, more generally, an XQuery expression. The XPath expression represented by variable **Path** is applied to all the values produced by the XQuery expression. According to the commented behavior, XQuery expressions can be unitary lists (**for**'s and XPath's) and non-unitary lists (**let**'s). The **xmlTag** constructor is also converted into a function **xmlTagX**:

```
xmlTagX :: string -> [xmlAttribute] -> XQuery -> XQuery
xmlTagX Name Attributes [Expr] =
  if Y == collect Expr then [xmlTag Name Attributes Y]
xmlTagX Name Attributes (X:Y:L) = [xmlTag Name Attributes (X:Y:L)]
```

Basically, this conversion is required to apply **collect** when either a **for** or an XPath expression provides the elements enclosed in an XML tag. With the previous definitions,  $\mathcal{TOY}$  accepts the following query:

```
R == xmlTagX "names" []
      (xLet X ("food.xml" <$-- name "food")
        'xReturn'
        xmlTagX "result" [] (X <$ (name "item" ./ .name "name")))
```

which simulates the query:

```
<names>
let $x:=doc("food.xml")/food return
<result> { $x/item/name } </result>
</names>
```

and outcomes the following answer:

```
{R -> [xmlTag "names" []
      [xmlTag "result" [] [
        xmlTag "name" [] [xmlText "watermelon" ],
        xmlTag "name" [] [xmlText "oranges" ],
        xmlTag "name" [] [xmlText "onions" ],
        xmlTag "name" [] [xmlText "strawberries" ]]]] }
```

It is worth noticing that  $\mathcal{TOY}$  shows not only the binding for  $R$ , but also for the variable  $X$ . If we are interested in the query without the values of the variables, we can introduce a function containing the code:

```
query = xmlTagX "names" []
      (xLet X ("food.xml" <$-- name "food")
        'xReturn'
        xmlTagX "result" [] (X <$ (name "item" ./ .name "name")))
```

and try the goal `query == R` to get the same result. In the case of `for` expressions, we can write:

```
query2 = xFor Y
      (xFor X ("food.xml" <$-- name "food")
        'xReturn' (X <$ (name "item" ./ .name "name")))
      'xReturn' Y
```

which simulates the following query:

```
for $Y in
(for $X in doc("food.xml")/food return $X/item/name)
return $Y
```

The following  $\mathcal{TOY}$  query returns four answers, once at a time, due to the use of non-determinism in the `for` expression:

```
Toy> query2== X
{ X -> [xmlTag "name" [] [xmlText "watermelon"]] }
{ X -> [xmlTag "name" [] [xmlText "oranges"]] }
{ X -> [xmlTag "name" [] [xmlText "onions"]] }
{ X -> [xmlTag "name" [] [xmlText "strawberries"]] }
```

## 4 XQuery Optimization in $\mathcal{TOY}$

In this section we present one of the advantages of using  $\mathcal{TOY}$  for running XQuery expressions. In [10] we have shown that XPath queries can be preprocessed by replacing the reverse axes by predicate filters including forward axes, as shown in [25]. In the case of XQuery, one of the optimizations to be achieved is to avoid XPath expressions at outermost positions. Here is an example of optimization. Consider the following query:

```
exam = xFor X
      (xFor Y ("food.xml" <$-- name "food" ./ . name "item")
        'xReturn'
        (xmlTagX "elem" []
          (xFor Z (Y <$ name "name")
            'xReturn' (xmlTagX "ids" [] Z))))
      'xReturn'
      (X <$ ((name "ids") ./ . (name "name")))
```

In such a query,  $(X <$ ((name "ids") ./ . (name "name")))$  is an XPath expression applied to an XML term constructed by the same query. By removing outermost XPath expressions, we can optimize XQuery expressions. In general, a place for optimization are nested XQuery expressions [21, 15]. In our case, we argue that XPath can be statically applied to XQuery expressions. The optimization comes from the fact that unnecessary XML terms can be built at run-time, and that removing them improves memory consumption. We observe in the previous query that "elem" and "ids" tags are useless, once we retrieve "name" from the original file. Therefore, the previous query can be rewritten into a more simpler and equivalent one:

```
examo = ("food.xml" <$-- name "food" ./ . name "item" ./ . name "name")
```

### 4.1 XQuery as Higher Order Patterns

In order to proceed with optimizations, we follow the same approach as in XPath. In [10] we have used the representation of XPath expressions for optimizing. As it was commented before, XPath operators are higher order operators, and then we can take advantage of the  $\mathcal{TOY}$  facilities for using higher order patterns to rewrite them. This is not the case, however, for XQuery expressions in  $\mathcal{TOY}$  because, for instance, `xFor` and `xLet` are always applied to two arguments, and therefore constitute reducible expressions, not higher-order patterns. In order to convert XQuery- $\mathcal{TOY}$  expressions into higher-order patterns, we propose a redefinition of the functions adding a dummy argument. Then, XQuery constructors can be redefined as follows:

```
yLet :: (A -> xQuery) -> (A -> xQuery) -> A -> xQuery
yLet X Y _ = xLet (X _) (Y _)
```

```
yFor :: (A -> xQuery) -> (A -> xQuery) -> A -> xQuery
yFor X Y _ = xFor (X _) (Y _)
```

The anonymous variable plays a role similar to the `quote` operator in Lisp [28]. In our case the expressions will become reducible when any extra argument is provided. In the meanwhile it can be considered as a data term, and as such it can be analyzed and modified. In the definitions above, `yLet` is reduced to `xLet` when such extra argument is provided. The two arguments `X` and `Y` also need their extra variable to become reducible. A variable is a special case, which has to be converted into a function (`xvar`):

```
xvar :: xQuery -> A -> xQuery
xvar X _ = X
```

Now, a given query can be rewritten as a higher order pattern. For instance, the previous `exam` can be represented as follows:

```
xexam = yFor (xvar X)
  (yFor (xvar Y) ("food.xml" <$$-- name "food" ./ .name "item")
    'yReturn'
      (xmlTagY "elem" []
        (yFor (xvar Z) ((xvar Y) <$$ (name "name"))
          'yReturn' (xmlTagY "ids" [] (xvar Z)) )))
    'yReturn'
      ((xvar X) <$$ ((name "ids") ./ . name "name")))
```

The query can be executed in *TOY* just providing any additional argument, in this case an anonymous variable:

```
Toy> xexam _ == R
{ R -> [xmlTag "name" [] [xmlText "watermelon" ] ] }
{ R -> [xmlTag "name" [] [xmlText "oranges" ] ] }
{ R -> [xmlTag "name" [] [xmlText "onions" ] ] }
{ R -> [xmlTag "name" [] [xmlText "strawberries" ] ] }
```

If the extra argument `_` is omitted, then the variable `R` is bound to the XQuery code `yFor (xvar X) (...name "name"))`. This behavior allows us to inspect and modify the query in the next subsection.

## 4.2 XQuery Transformations

Now, we would like to show how to rewrite XQuery expressions in order to optimize them. We have defined a set of transformation rules for removing outermost XPath expressions, when possible. Let us remark that correctness of the transformation rules, that is, preserving equivalence, is out of the scope of this paper. An example of (a subset of) the transformation rules is:

```

reduce ((yFor (xvar Z) E) 'yReturn' (xvar Z)) = E
reduce ((xmlTagY N A E) <$$ P) = reduce_xml (xmlTagY N A E) P
reduce_xml (xmlTagY N A E) P = reduce_xmlPath E P
reduce_xmlPath (xmlTagY N A E) P =
    if P == (name N) ./ . P2
    then reduce_xmlPath E P2
    else ((xmlTagY N A E) <$$ P)
...

```

The first reduction rule removes the unnecessary `for` expressions that define a variable `Z` taking a value `E` only to return `Z`. The second rule removes XPath expressions that traverse elements built in the same query. For instance, an expression of the form `$X/a/b`, with `$X` of the form `<a>E</a>` is reduced to `$Y/b` with `$Y/b` and `$X` taking the value `E` (this transformation is performed by function `reduce_xmlPath`). The optimizer can be defined as the fixpoint of function `reduce`:

```

optimize :: (A -> xQuery) -> (A -> xQuery)
optimize X = iterate reduce X

iterate :: (A -> A) -> A -> A
iterate G X = if Y == X then Y else iterate G Y
              where Y = (G X)

```

For instance, the running example is optimized as follows:

```

Toy> optimize xexam == X
{ X -> (<$$-- "food.xml" child ... (nameT "food") ./
      child ... (nameT "item") ./ child ... (nameT "name")) }

```

Finally, an XQuery expression is executed (with optimizations) in  $\mathcal{TOY}$  by calling the function `run`, which is defined as:

```

run :: (A -> xQuery) -> xQuery
run X = (optimize X) _

```

By using `run`,  $\mathcal{TOY}$  obtains the same four answers as with the original query:

```

Toy> run xexam == X
{ R -> [xmlTag "name" [] [xmlText "watermelon" ] ] }
{ R -> [xmlTag "name" [] [xmlText "oranges" ] ] }
{ R -> [xmlTag "name" [] [xmlText "onions" ] ] }
{ R -> [xmlTag "name" [] [xmlText "strawberries" ] ] }

```

In order to analyze the performance of the optimization, the next table compares the elapsed time for the query running on  $\mathcal{TOY}$  before and after the optimization, with respect to different sizes for file `"food.xml"`.

<i>Items</i>	<i>Initial Query</i>	<i>Optimized Query</i>	<i>Speed-up</i>
1,000	1.9	0.4	4.8
2,000	3.7	0.8	9.3
4,000	7.4	1.7	4.4
8,000	18.1	3.9	4.6
16,000	36.0	7.8	4.6

The first column indicates the number of `item` elements included in "`food.xml`", the second and third column display the time in seconds required by the original and the optimized query, respectively, and the last column displays the speed-up of the optimized code. In order to force the queries to find all the answers, the submitted goals are `(exam == R, false)` and `(run xexam == R, false)`, corresponding to the initial and the optimized query, respectively. The atom `false` after the first atomic subgoal always fails, forcing the reevaluation until no more solutions exist. As can be seen in the table, in this experiment the optimized query is above 4.5 times faster in the average than the initial one. In other experiments (for instance, replacing `for` by `let` in this example) the difference can be noticeable also in terms of memory, since the system runs out of memory computing the query before optimization, but works fine with the optimized query. Of course, more extensive benchmarks would be needed to assess this preliminary results. However, the purpose of this paper is not to propose or to evaluate XQuery optimizations, but to show how they can be easily incorporated and tested in our framework.

## 5 Conclusions

We have shown how the declarative nature of the XML query language XQuery fits in a very natural way in functional-logic languages. Our setting fruitfully combines the collection of results required by XQuery `let` statements and the use of individual values as required by `for` statements and XPath expressions. For the users of the functional-logic  $\mathcal{TOY}$ , the advantage is clear: they can use queries very similar to XQuery in their programs. Although adapting to the  $\mathcal{TOY}$  syntax can be hard at first, we think that the queries are close enough to their equivalents in native XQuery. However, we would like to go further by providing a parser from XQuery standard syntax to the equivalent  $\mathcal{TOY}$  expressions.

From the point of view of the XQuery apprentices, the tool can be useful, specially if they have some previous knowledge of declarative languages. The possibility of testing query optimizations can be very helpful. The paper shows a technique based on the use of additional dummy variables for converting queries in higher-order patterns. A similar idea would be to use a data type for representing the query and then a parser/interpreter for evaluating this data type. However, we think that the approach considered here has a higher abstraction level, since the queries can not only be analyzed, they can also be computed by simply providing an additional argument. Finally, the framework can also



be interesting for designers of XQuery environments, because it allows users to easily define prototypes of new features such as new combinators and functions.

A version of the *TOY* system including the examples of this paper can be downloaded from <http://gpd.sip.ucm.es/rafa/wflp2011/toyxquery.rar>

## References

1. Almendros-Jiménez, J., Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Declarative Embedding of XQuery in a Functional-Logic Language. Technical Report SIC-04/11, Facultad de Informática, Universidad Complutense de Madrid (2011), <http://gpd.sip.ucm.es/rafa/xquery/>
2. Almendros-Jiménez, J.M.: An Encoding of XQuery in Prolog. In: Bellahsene, Z., Hunt, E., Rys, M., Unland, R. (eds.) XSym 2009. LNCS, vol. 5679, pp. 145–155. Springer, Heidelberg (2009)
3. Almendros-Jiménez, J.M., Becerra-Terón, A., Enciso-Baños, F.J.: Querying XML documents in logic programming. *Journal of Theory and Practice of Logic Programming* 8(3), 323–361 (2008)
4. 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)
5. 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)
6. 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)
7. Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 479–490. ACM Press, New York (2006)
8. 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)
9. Bry, F., Schaffert, S.: The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In: Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R. (eds.) NODe-WS 2002. LNCS, vol. 2593, pp. 295–310. Springer, Heidelberg (2003)
10. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Integrating XPath with the Functional-Logic Language Toy. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 145–159. Springer, Heidelberg (2011)
11. Caballero, R., López-Fraguas, F.: A functional-logic perspective on parsing. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 85–99. Springer, Heidelberg (1999)
12. 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)
13. 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)
14. Fegaras, L.: HXQ: A Compiler from XQuery to Haskell (2010)

15. Grinev, M., Pleshachkov, P.: Rewriting-based optimization for XQuery transformational queries. In: 9th International Database Engineering and Application Symposium, IDEAS 2005, pp. 163–174. IEEE Computer Society, Los Alamitos (2005)
16. Guerra, R., Jeuring, J., Swierstra, S.D.: Generic validation in an XPath-Haskell data binding. In: Proceedings Plan-X (2005)
17. Hanus, M.: Curry: An Integrated Functional Logic Language (2003), <http://www.informatik.uni-kiel.de/~mh/curry/> (version 0.8.2 March 28, 2006)
18. Hanus, M.: Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel (2011)
19. Hosoya, H., Pierce, B.C.: XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology* 3(2), 117–148 (2003)
20. Hutton, G., Meijer, E.: Monadic parsing in Haskell. *J. Funct. Program.* 8(4), 437–444 (1998)
21. Koch, C.: On the role of composition in XQuery. In: Proc. WebDB (2005)
22. Fraguas, F.J.L., Hernández, J.S.: TOY: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
23. Marian, A., Simeon, J.: Projecting XML Documents. In: Proc. of International Conference on Very Large Databases, pp. 213–224. Morgan Kaufmann, Burlington (2003)
24. 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)
25. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking forward. In: Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W. (eds.) EDBT 2002. LNCS, vol. 2490, pp. 109–127. Springer, Heidelberg (2002)
26. Ronen, R., Shmueli, O.: Evaluation of datalog extended with an XPath predicate. In: Proceedings of the 9th Annual ACM International Workshop on Web Information and Data Management, pp. 9–16. ACM, New York (2007)
27. 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. CEUR Workshop Proceedings, vol. 60, p. 22 (2002)
28. Seibel, P.: Practical Common Lisp. Apress (2004)
29. Seipel, D.: Processing XML-Documents in Prolog. In: Procs. of the Workshop on Logic Programming 2002, p. 15. Technische Universität Dresden, Dresden (2002)
30. Sulzmann, M., Lu, K.Z.: XHaskell – adding regular expression types to haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 75–92. Springer, Heidelberg (2008)
31. Thiemann, P.: A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming* 12(4&5), 435–468 (2002)
32. W3C. XML Schema 1.1
33. W3C. Extensible Markup Language, XML (2007)
34. W3C. XML Path Language (XPath) 2.0 (2007)
35. W3C. XQuery 1.0: An XML Query Language (2007)
36. 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)
37. Walmsley, P.: XQuery. O’Reilly Media, Inc., Sebastopol (2007)
38. Wielemaker, J.: SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam (March 2005)

# Size Invariant and Ranking Function Synthesis in a Functional Language<sup>\*</sup>

Ricardo Peña and Agustin D. Delgado-Muñoz

Universidad Complutense de Madrid, Spain  
ricardo@sip.ucm.es, elsmda@gmail.com

**Abstract.** Size analysis is concerned with the compile-time determination of upper bounds to the size of program variables, including the size of the results returned by functions. It is useful in many situations and also as a prior step to facilitate other analyses, such as termination proofs. Ranking function synthesis is one way of proving termination of loops or of recursive definitions.

We present a result in automatic inference of size invariants, and of ranking functions proving termination of functional programs, by adapting linear techniques developed for other languages. The results are accurate and allow us to solve some problems left open in previous works on automatic inference of safe memory bounds.

**Keywords:** functional languages, linear techniques, abstract interpretation, size analysis, ranking functions.

## 1 Introduction

Size analysis allows us to determine the individual size of (some) program variables, or more commonly, the size relationship between groups of them. In the latter case, they are called *size invariants*. Size analysis has been used in the past to help the memory management system to estimate the size of data structures at compile time [13], or to obtain inter-argument size relationships in predicates of logic programs [7]. It is also useful for providing size variations of arguments in recursive procedures, thus facilitating building termination proofs.

In our case, the motivation for designing a size analysis is using it as a previous step in determining the memory consumption of functional programs written in our language *Safe*. In a prior work [18] we presented some static analysis based algorithms for inferring upper bounds to memory consumption of programs. The technique used was abstract interpretation, using the complete lattice of monotonic functions  $(\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$  ordered by point-wise  $\leq$  as abstract domain, where  $n$  is the number of arguments of the function being analysed. We showed some examples of applications, and the bounds obtained were good in simple programs. For instance, we got precise linear heap and stack bounds for list functions such as *merge*, *append*, and *split*, and quadratic over-approximations for the heap

---

<sup>\*</sup> Work partially funded by the projects TIN2008-06622-C03-01/TIN (STAMP), and S2009/TIC-1465 (PROMETIDOS).

consumption of functions such as *mergesort* and *quicksort*. The algorithms were even able to infer a constant stack space for tail recursive functions.

A remarkable feature of the algorithms was that in some circumstances the abstract interpretation was *reductive* in the lattice meaning that, by introducing as hypothesis the previously inferred bound, and by iterating the interpretation, a tighter bound and still a correct one could be obtained.

Unfortunately, the work was incomplete because it needed some information to be introduced by hand for every particular program, namely the size of some local variables and an upper bound to the length of the longest call chain of recursive functions. These two problems deserve independent and complex analyses by themselves, and we decided to defer their solution. The algorithms were proved correct, provided correct functions for these figures were given.

In a prior paper [15] we approached one of these problems — inferring the length of the longest recursive call chain — by translating our functional programs into a term rewriting system (TRS) and then using termination proofs of TRSs, based on dependency pairs and polynomial synthesis, for computing this bound. The first results were encouraging but the approach could not prove any bound for simple algorithms such as *mergesort* and *quicksort*. We felt that having a previous size analysis could probably improve the termination proofs.

In this paper we approach both size analysis and termination proofs by using linear techniques, which have proved successful in analysing imperative and logic languages, and even bytecode programs. We do not know of any relevant application of these techniques to functional languages.

The main contribution of the paper is showing that these techniques have for first-order functional languages the same power as for any other paradigm. The adaptations needed essentially consist of applying some transformations to the source programs, and taking some care when applying the techniques in a context different to the one they were conceived in.

The plan of the paper is as follows: In Sec. 2 we do a brief survey of linear techniques applied to both size analysis and ranking function synthesis. Then, Sec. 3 presents the aspects of our functional language *Safe* that are relevant to this paper. Sec. 4 is devoted to obtaining size invariants of *Safe* programs, while Sec. 5 adapts the well-known ranking function synthesis method for loops by Podelski and Rybalchenko [19] to computing our bound to the longest call chain. Finally, Sec. 6 provides a short conclusion.

## 2 Linear Constraints Techniques

Abstract interpretation [10] is a very powerful static analysis technique able to infer a variety of properties in programs written in virtually any programming language. In functional languages it has been successfully applied to strictness and update avoidance analyses of lazy languages, to sharing and binding time analyses, and many others. The abstract domains are usually finite small ones when abstracting non-functional values, but they tend to grow exponentially when they are extended to higher-order values.

Polyhedral abstract domains have been extensively used in logic and imperative languages, but not so frequently in functional ones. These domains are useful when quantitative rather than qualitative properties are sought for, as it is the case of size or cost relations between variables. Since the seminal work of Cousot and Halbwachs [11], polyhedra have been used to analyse arithmetic linear relations between program variables. A convex polyhedron is a subset of  $\mathbb{R}^n$  which is the intersection of a finite number of semi-spaces. There exist at least two finite representations of convex polyhedra:

- By three sets, namely of vertexes, rays, and lines in a  $n$ -dimensional space. Rays and lines are needed when the polyhedron is unbounded in order to specify in which directions it extends to infinity.
- By a conjunction of linear constraints between  $n$  variables. Each constraint determines a semi-space.

There are algorithms for translating these representations into each other, although their cost is exponential in time. A frequent (and also costly) operation is to compute the convex hull of two polyhedra, which is the minimum convex polyhedron containing both. This operation is associative and commutative. The advantage of using linear constraints is that most of the interesting problems involving them are decidable. For instance: to know whether a set of constraints is satisfiable; whether a constraint is implied by a set of other ones; to project a set of constraints over a subset of their variables; to compute the convex hull of two sets of constraints; to maximise (minimise) a linear function with respect to a set of constraints; and others.

*Invariant synthesis.* In the context of imperative languages, an invariant is a linear relation between the variables involved in a loop, holding at the beginning of each iteration. An abstract interpretation for synthesising loop invariants starts by computing a polyhedron with the relations known at the beginning of the loop, and iterates calculating the convex hull between the polyhedron coming from the previous iteration and the one obtained by the transition relation of the loop body. After a few iterations, some relations stabilise while some others do not. The former constitute the invariant. Several tools have been developed for obtaining these invariants, (for instance ASPIC, see [12]), or for giving the necessary infrastructure to compute them (cf. [3]).

In the logic programming field, Benoy and King [7] applied a similar technique to the inference of size relations between the arguments of a logic predicate. As a first step, the logic program is transformed into an abstract one on arithmetic variables, by replacing the original predicate arguments by their corresponding sizes. An abstract interpretation of the transformed program infers the invariant size relations between the arguments of the original program. The ascending chain (in the sense of set inclusion) of polyhedra obtained by the fixpoint algorithm may in principle be infinite, because some relations do not stabilise. A *widening* technique [14] is used to eliminate these variant relations while the invariant ones are retained. Of course, if the invariant relations are not linear the algorithm does not obtain anything meaningful.

*Ranking function synthesis.* Detecting termination statically has attracted the attention of much research work. Given that this is an undecidable problem in general, the algorithms try to cover as many decidable cases as possible. One successful approach has been the work by Ben-Amram and his group, starting in the seminal paper [14], where in a first phase the program being analysed is transformed into a so-called *size-change graph*. This is the program call flow graph enriched with information about which arguments strictly decrease at a call and which ones may decrease or remain equal. This part of the analysis is outside of the proposed termination algorithms, and may be done by hand or by a previous size analysis. Nice about this approach is that termination of size-change graphs is decidable, although the algorithm is exponential in the worst case (these programs are called *size-change terminating*, or SCT). However, by using benchmarks the authors convincingly show that this case is very unusual and that most of the time a polynomial algorithm suffices [6]. Moreover, in many cases they can synthesise a global ranking function ensuring that the program terminates, which can be proved decreasing at each transition in polynomial time. Synthesising such a function is however a NP-complete problem which they decide by using a SAT-solver [5]. Their algorithm is complete for SCT problems admitting a global ranking function of a certain shape (lexicographic tuples of some simple well-founded relations called *level mappings*).

Another successful line of research has been the synthesis of linear ranking functions. Podelski and Rybalchenko give in [19] a complete method to synthesise this kind of function for simple while-loops in which the loop guard is a conjunction of linear expressions, and the loop body is a multiple assignment of linear expressions to variables. The kernel of the method is solving a set of linear constraints and it will be explained in more detail in Sec. 5. This small piece can be the basis for inferring termination of more complex programs. In [20] they define that a relation is *disjunctively well-founded* if it is the union of a number of well-founded relations (which in general is not well-founded). They completely characterise program termination by the existence of such a relation if at the same time it is an invariant of the transition relation. The idea they try to exploit is proving global termination of programs with nested loops by associating a simple well-founded relation to each of the loops, and proving that their union is a transition invariant.

Another method for analysing complex loop structures and synthesising linear ranking functions is by Colon and Sipma in [9], which needs a prior size analysis for computing linear loop invariants. The authors do not claim that the method is complete for such kind of ranking functions. A recent work extending Colon and Sipma’s approach is [2]. It presents a complete method for synthesising a global ranking function for a complex nested control flow structure, provided the ranking function has the form of a lexicographically ordered tuple of linear expressions.

Albert *et al* claim in [1] — although not many details are given — to have used Podelski and Rybalchenko’s method as one of the steps for solving recurrence relations obtained by analysing Java bytecode programs. The idea is to use the

$prog \rightarrow \overline{data_i}; \overline{dec_j}; e$	{Core-Safe program}
$dec \rightarrow f \overline{x_i} = e$	{recursive, polymorphic function definition}
$e \rightarrow a$	{atom $a$ : either a literal $c$ or a variable $x$ }
$  a_1 \oplus a_2$	{primitive operator application}
$  f \overline{a_i}$	{function application}
$  C \overline{a_i}$	{constructor application}
$  \mathbf{let} \ x_1 = e_1 \ \mathbf{in} \ e_2$	{non-recursive, monomorphic <b>let</b> }
$  \mathbf{case} \ x \ \mathbf{of} \ \overline{alt_i}$	{case expression}
$alt \rightarrow C \overline{x_i} \rightarrow e$	{case alternative}

**Fig. 1.** Simplified *Core-Safe* syntax

ranking function as an upper bound to the recurrence depth (i.e. to the number of unfoldings required in the worst case to reach a non-recursive case). We will pursue this idea here for inferring an upper bound to the depth of the call tree of a recursive *Safe* function.

### 3 The *Safe* Functional Language

*Safe* is a first-order eager polymorphic functional language with a syntax similar to that of (first-order) Haskell, and featuring an unusual memory management system. Its main aim is to facilitate the compile-time inference of safe upper bounds to memory consumption. Its memory model is based on disjoint heap regions where data structures are built. The compiler infers the optimal way to assign data to regions, so that the data lifetimes are as short as possible while compatible with allocating and deallocating regions by following a stack-based strategy. The region-based model has two benefits:

- A garbage collector is not needed.
- The compiler may compute an upper bound to the size of each region and then to the whole heap.

More information about *Safe*, its type system, and its memory inference algorithms can be found at [\[16,17,18\]](#).

The *Safe* front-end desugars *Full-Safe* and produces a bare-bones functional language called *Core-Safe*. The transformation starts with region inference and goes on with Hindley-Milner type inference, desugaring pattern matching into **case** expressions, transforming **where** clauses into **let** expressions, collapsing several function-defining equations into a single one, and some other simple transformations.

As regions are not relevant to this paper, in Fig. 1 we show a simplified *Core-Safe*'s syntax where regions have been omitted. A program *prog* is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression *e* whose value is the program result. The abbreviation  $\overline{x_i}$  stands for  $x_1 \cdots x_n$ , for some  $n$ .

```

split 0 ys      = ([], ys)
split n []      = ([], [])
split n (y:ys)  = (y:ys1,ys2)   where (ys1, ys2) = split (n-1) ys

merge [] ys     = ys
merge xs []     = xs
merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys

msort []        = []
msort [x]       = [x]
msort xs        = merge (msort xs1) (msort xs2)
                  where (xs1,xs2) = split (length n / 2) xs

```

**Fig. 2.** *mergesort* algorithm in *Full-Safe*

```

merge x y = case x of
  []      -> y
  ex:x'   -> case y of
    []     -> x
    ey:y'  -> case ex <= ey of
      True  -> let z1 = merge x' y in ex:z1
      False -> let z2 = merge x y' in ey:z2

msort x = case x of
  []      -> []
  ex:x'   -> case x' of
    []     -> ex:[]
    _:_    -> let n  = length x      in
               let n2 = n/2          in
               let (x1,x2) = split n2 x in
               let z1 = msort x1      in
               let z2 = msort x2      in
               merge z1 z2

```

**Fig. 3.** functions *merge* and *msort* in *Core-Safe*

In Fig. 2 we show a *Full-Safe* version of the *mergesort* algorithm, which we will use as running example throughout the paper. In Fig. 3 we show the translation to *Core-Safe* of two of its functions.

Our purpose is to analyse *Core-Safe* programs to infer invariant size relations between the arguments and results of each function, and upper bounds to the runtime size of the call tree unfolded when invoking a recursive function. As part of this process, it will be important to discover which sub-expressions contribute to the non-recursive (or base) cases of a recursive function, and which ones contribute to the recursive ones. Moreover, we will distinguish between mutually exclusive recursive calls, i.e. those which will *never* execute together at runtime



```

data  $QExp = B\ Exp \mid R\ Exp \mid Var := [QExp]$ 

 $seqs_f :: Exp \rightarrow [[QExp]]$ 
 $seqs_f\ e = [[B\ e]]$  -- if  $e \in \{c, x, a_1 \oplus a_2, g\ \overline{a_i}, C\ \overline{a_i}\}$ 
 $seqs_f\ (f\ \overline{a_i}) = [[R\ (f\ \overline{a_i})]]$ 
 $seqs_f\ (\mathbf{case\ of\ alts}) = concat\ [seqs_f\ e \mid (C\ \overline{x_i} \rightarrow e) \in alts]$ 
 $seqs_f\ (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2) = [(x_1 := s_1) : s_2 \mid s_1 \in seqs_f\ e_1, s_2 \in seqs_f\ e_2]$ 

```

**Fig. 4.** Algorithm for extracting the base and recursive cases of a *Core-Safe* expression

```

 $seqs_{merge} = [[B\ y], [B\ x],$ 
 $\quad [z_1 := [R\ (merge\ x'\ y)], B\ (e_x : z_1)], [z_2 := [R\ (merge\ x\ y')], B\ (e_y : z_2)]]$ 

 $seqs_{msort} = [[B\ []], [B\ (e_x : [])],$ 
 $\quad [n := [B\ (length\ x)], n_2 := [B\ (n/2)], (x_1, x_2) := [B\ (split\ n_2\ x)],$ 
 $\quad z_1 := [R\ (msort\ x_1)], z_2 := [R\ (msort\ x_2)], B\ (merge\ z_1\ z_2)]]$ 

```

**Fig. 5.** Decomposition of *merge* and *msort* into base and recursive cases

(e.g. those occurring in different alternatives of a **case** expression), and those with *may* execute one after the other, i.e. potentially sequential recursive calls.

An approximation to this property can be obtained by an algorithm separating textual calls occurring in different branches of a **case**, and putting together textual calls occurring in the sub-expressions  $e_1$  and  $e_2$  of a **let**. In Fig. 4 we show the algorithm  $seqs_f$ , written in a Haskell-like language, analysing this property for the *Core-Safe* expression corresponding to function- $f$ 's body. It returns a list of lists of *qualified* expressions, meaning by this that a tag  $B$  (for base cases), or  $R$  (for recursive ones) is added to each individual sub-expression. Each internal list represents a mutually exclusive way of executing the function (i.e. a potential *path* through its body), while the sub-expressions inside a list indicate a sequential execution of them.

In Fig. 5 we show the application of the algorithm to *merge* and *msort*. For *merge*  $x\ y$  we obtain four internal lists illustrating that there are two mutually exclusive base cases, and that the two recursive calls exclude each other. When applied to *msort*( $x$ ) we obtain three internal lists illustrating that there are two base cases, and that the two recursive calls may be sequentially executed.

## 4 Size Invariant Inference

Following [7], in order to reason about sizes, the original program must first be translated into an abstract program in which data structures have been replaced by their corresponding sizes, and previously to that, a notion of *size* must be defined. For logic programs, the more frequently used ones are the list length for list arguments, the value for integer arguments, zero for any other atom, and the term size for the rest of variables.

$$\begin{aligned}
\text{merge}^S x y = & \text{case } x \text{ of} \\
& x = 1 \rightarrow y \\
& x \geq 2 \rightarrow \text{case } y \text{ of} \\
& \quad y = 1 \rightarrow x \\
& \quad y \geq 2 \rightarrow \text{case } ? \text{ of} \\
& \quad \quad T \rightarrow \text{let } z_1 = \text{merge}^S (x - 1) y \text{ in } z_1 + 1 \\
& \quad \quad F \rightarrow \text{let } z_2 = \text{merge}^S x (y - 1) \text{ in } z_2 + 1 \\
\text{msort}^S x = & \text{case } x \text{ of} \\
& x = 1 \rightarrow 1 \\
& x \geq 2 \rightarrow \text{case } x \text{ of} \\
& \quad x = 2 \rightarrow 2 \\
& \quad x \geq 3 \rightarrow \text{let } n = \text{length}^S x \text{ in} \\
& \quad \quad \text{let } n_2 = n/2 \text{ in} \\
& \quad \quad \text{let } (x_1, x_2) = \text{split}^S n_2 x \text{ in} \\
& \quad \quad \text{let } z_1 = \text{msort}^S x_1 \text{ in} \\
& \quad \quad \text{let } z_2 = \text{msort}^S x_2 \text{ in} \\
& \quad \quad \text{merge}^S z_1 z_2
\end{aligned}$$

**Fig. 6.** Abstract size functions *merge* and *msort*

In our memory model we have a precise notion of size in terms of memory cells occupied by a data structure: each constructor application fits exactly in one cell. So, the size of a list is its length plus one because the empty list constructor needs an additional cell. Moreover, we have a precise notion of *data structure*: it comprises the set of cells corresponding to its recursive spine. For instance, a list of  $n$  lists constitutes  $n + 1$  independent data structures, while a binary tree of integers is just one. Additionally we define:

- The size of an integer constant or variable is its value  $n$ .
- The size of a Boolean constant or variable is zero.

We will call *size programs*, and *size functions*, to the abstract programs and functions resulting from the size translation. If  $f$  is the original function,  $f^S$  will denote its size version. The size functions resulting from the translation of *merge* and *msort* of Fig. 3 are shown in Fig 6.

The next step consists of performing an abstract interpretation of the size functions. The abstract domain will be that of convex polyhedra ordered by set inclusion, represented in this paper by conjunctions of linear constraints. The *meet* operation, or greatest lower bound  $\sqcap$ , is the intersection of two polyhedra, and consists of just putting together the two sets of constraints. The *join* operation, or least upper bound  $\sqcup$ , is the convex hull of the two polyhedra. We have used the algorithm developed by Andy King *et al* [8] to compute the convex hull of two polyhedra represented by sets of linear constraints, and giving as result another constraint set.

The algorithm we propose has been inspired by Benoy and King’s algorithm [7] for analysing logic programs. It consists of the following steps:

*Analysing a complete Core-Safe program*

1. The size functions obtained by translating a *Core-Safe* program are analysed in the order they occur in the file: from low-level ones not using any other function, to higher-level ones using those previously defined in the file.
2. For each size function  $g^S$ , a set of invariant size relations between input arguments and output results are inferred.
3. These relations are kept in a global environment  $\Sigma$ . When analysing the current function, say  $f^S$ , calling an already analysed function  $g$ , the invariant relations of the latter are obtained from  $\Sigma$  and instantiated with the sizes of the actual arguments  $\overline{a_i}^n$  used in the calls to  $g$ . These relations, together with the rest of relations inferred for  $f^S$ , are used to compute  $f^S$ 's invariant relations.

*Analysing a function: extracting the constraints* Analysing the current function  $f^S$  consists of the following steps. In order to fix ideas, we will call  $\overline{x_i}$  to  $f^S$ 's formal arguments and  $z$  to its result (or  $\overline{z_j}$  if the result is a tuple).

1. Function  $seqs_f$  of Fig. 4 is applied to function- $f^S$ 's body in a similar way as it was applied to a *Core-Safe* expression.
2. As a result, sets of constraints, one set for each of the code sequences, are inferred. The base sequences constraint sets are then separated from the recursive ones (recursive sequences can be identified by the presence of at least a recursive call to  $f^S$ ).
3. The constraints of each base sequence are expressed in terms of  $\overline{x_i}$  and  $\overline{z_j}$ . This can always be done by projecting a set of constraints with more variables to variables  $\overline{x_i}$  and  $\overline{z_j}$ .
4. The constraints of each recursive sequence are expressed in terms of  $\overline{x_i}$ ,  $\overline{z_j}$ , and of two sets of variables  $\overline{x_i}^k$  and  $\overline{z_j}^k$  for each recursive call  $k$  in the sequence. The variables  $\overline{x_i}^k$  represent the input arguments sizes of that call, while the  $\overline{z_j}^k$  represent its output sizes.

*Analysing a function: fixpoint iteration* Then, a fixpoint algorithm for  $f^S$  is launched having the following steps:

1. The initial polyhedron is the convex hull

$$P_{next} = B_1 \sqcup \dots \sqcup B_n$$

being  $B_l(\overline{x_i}, \overline{z_j})$ ,  $1 \leq l \leq n$ , the polyhedra defined by the sets of constraints of the base cases.

2. At each iteration, the variables of polyhedron  $P_{next}(\overline{x_i}, \overline{z_j})$  are renamed, obtaining  $Q_{prev} = P_{next}[\overline{x_i}'/\overline{x_i}, \overline{z_j}'/\overline{z_j}]$ . The idea is that  $Q_{prev}(\overline{x_i}', \overline{z_j}')$  represents the constraints coming from the previous iterations.
3. Now, for each recursive sequence  $l$ , its constraints are enriched by adding the constraints coming from  $Q_{prev}$ , as many times  $n_l$  as recursive calls are

$$\begin{aligned}
B_1^{merge} &= \{x = 1, z = y\} \\
B_2^{merge} &= \{x \geq 2, y = 1, z = x\} \\
R_1^{merge} &= \{x \geq 2, y \geq 2, x' = x - 1, y' = y, z = 1 + z'\} \\
R_2^{merge} &= \{x \geq 2, y \geq 2, x' = x, y' = y - 1, z = 1 + z'\} \\
\\ 
B_1^{msort} &= \{x = 1, z = 1\} \\
B_2^{msort} &= \{x = 2, z = 2\} \\
R_1^{msort} &= \{x \geq 3, x + 1 = x'_1 + x'_2, z + 1 = z'_1 + z'_2\}
\end{aligned}$$

**Fig. 7.** Restrictions corresponding to the base and recursive cases of *merge* and *msort*

in the sequence, by previously substituting the  $\overline{x'_i}$  for the  $\overline{x_i^k}$  and the  $\overline{z'_j}$  for the  $\overline{z_j^k}$ ,  $1 \leq k \leq n_l$ . Let us call

$$R_l(\overline{x_i}, \overline{z_j}, \overline{x_i^1}, \overline{z_j^1}, \dots, \overline{x_i^{n_l}}, \overline{z_j^{n_l}})$$

to the polyhedron resulting from the  $l$ -th sequence. In geometric terms, each polyhedron  $R_l$  represents the intersection of the polyhedra determined by the constraints coming from the recursive case  $l$  with the one determined by the constraints coming from the internal calls of this sequence.

4. Each  $R_l$  is projected over the variables  $\overline{x_i}, \overline{z_j}$  obtaining  $RP_l(\overline{x_i}, \overline{z_j})$ . If there are  $m$  recursive sequences, then the following convex hull is computed:

$$P_{next}(\overline{x_i}, \overline{z_j}) = RP_1 \sqcup \dots \sqcup RP_m \sqcup B_1 \sqcup \dots \sqcup B_n$$

5. If  $P_{next}[\overline{x'_i}/\overline{x_i}, \overline{z'_j}/\overline{z_j}] = Q_{prev}$  then stop; else go to (2).

## Two examples

In Fig. 7 we show the base and recursive sets of constraints inferred for *merge* and *msort* before launching the fixpoint algorithm. In the *merge* case, the constraints are easily obtained from the sequences resulting from  $seqs_{merge} (merge^S)$ . In the *msort* case, the following relations obtained from the *length*, *split* and *merge* invariants, are additionally needed:

$$\begin{array}{ll}
n + 1 = x & n := \text{length } x \\
n_2 = x \text{ div } 2 & n_2 := n/2 \\
x + 1 = x_1 + x_2, x \geq x_2, x \leq n_2 + x_2 & (x_1, x_2) := \text{split } n_2 \ x \\
z + 1 = z_1 + z_2 & z := \text{merge } z_1 \ z_2
\end{array}$$

In Fig. 8 we show the polyhedra obtained for *merge* and *msort* after a few iterations of the fixpoint algorithm. For *merge* the fixpoint is reached after the first iteration, obtaining as invariant  $\{x \geq 1, z + 1 = x + y\}$ . For *msort*, the ascendant chain is infinite because of the restrictions  $x \leq 2, x \leq 3, x \leq 4, \dots$ . The widening technique used in [7], and original from [11], eliminates this just by keeping as  $P_{next}$  the restrictions of iteration  $i$  implied by the ones of iteration  $i + 1$ . This leads to  $\{x \geq 1, z = x\}$  as the size invariant of *msort*. In both cases, the invariants precisely describe the size relations between the input and the output lists.

Iter.	$\mathbf{P}_{next}$	$\mathbf{R}_i$
$1_{merge}$	$\{x \geq 1, z + 1 = x + y\}$	$R_1 = \{x \geq 2, y \geq 2, x' = x - 1, y' = y, z = 1 + z', x' \geq 1, z' + 1 = x' + y'\}$ $R_2 = \{x \geq 2, y \geq 2, x' = x, y' = y - 1, z = 1 + z', x' \geq 1, z' + 1 = x' + y'\}$
$2_{merge}$	$\{x \geq 1, z + 1 = x + y\}$	
$1_{msort}$	$\{x \geq 1, x \leq 2, z = x\}$	$R_1 = \{x \geq 3, x + 1 = x'_1 + x'_2, z + 1 = z'_1 + z'_2, x'_1 \geq 1, x'_1 \leq 2, z'_1 = x'_1, x'_2 \geq 1, x'_2 \leq 2, z'_2 = x'_2\}$
$2_{msort}$	$\{x \geq 1, x \leq 3, z = x\}$	$R_1 = \{x \geq 3, x + 1 = x'_1 + x'_2, z + 1 = z'_1 + z'_2, x'_1 \geq 1, x'_1 \leq 3, z'_1 = x'_1, x'_2 \geq 1, x'_2 \leq 3, z'_2 = x'_2\}$
$3_{msort}$	$\{x \geq 1, x \leq 4, z = x\}$	

**Fig. 8.** Fixpoint iterations for *merge* and *msort*

## 5 Ranking Function Synthesis

In [18] we presented three inference algorithms to statically obtain upper bounds to the memory consumption of a *Core-Safe* recursive function. The first one was for inferring the *live* heap memory contributed by a call to the function. The second one for inferring the *peak* heap memory needed by the function, and a last one for inferring its peak stack memory. There, we used several functions assumed to be assumed correct for getting the following data:

- $nr_f(\bar{x})$  and  $nb_f(\bar{x})$  respectively gave upper bounds to the number of non-base and base calls of the runtime call tree unfolded by function  $f$  when it is called with arguments sizes  $\bar{x}$ . A non-base call is one recursively calling  $f$  again, and a base call is one ending a chain of recursive calls to  $f$ .
- $len_f(\bar{x})$  gave an upper bound to the length of the longest chain of recursive calls to  $f$ .
- $|y|_f(\bar{x})$  gave an upper bound to the size of variable  $y$  (assumed to belong to  $f$ 's body) as a function of  $f$ 's argument sizes  $\bar{x}$ .

The algorithms were proved correct assuming that we had correct functions for the above figures, but we left open how to infer them. A first step for inferring size functions  $|y|_f$  has been given with the inference of size invariants presented in Sec. 4. By following a similar idea to [11],  $nr_f$  and  $nb_f$  can be approximated should we have a correct function for  $len_f(\bar{x})$ . In effect, having  $len_f(\bar{x})$  and a bound  $n_f$  to the maximum number of calls issued from an invocation to  $f$ , we can compute the above functions as follows:

$$nb_f(\bar{x}) = \begin{cases} 1 & \text{if } n_f = 1 \\ n_f^{len_f(\bar{x})-1} & \text{if } n_f > 1 \end{cases}$$

$$nr_f(\bar{x}) = \begin{cases} len_f(\bar{x}) - 1 & \text{if } n_f = 1 \\ \frac{n_f^{len_f(\bar{x})-1} - 1}{n_f - 1} & \text{if } n_f > 1 \end{cases}$$

This figures correspond to the internal and leaf nodes of a complete tree of branching factor  $n_f$  and height  $len_f(\bar{x})$ . The branching factor  $n_f$  is a static quantity easily computed by taking the maximum number of consecutive calls in the sequences returned by function  $seqs_f$  of Fig. 4. For instance,  $n_{merge} = 1$  and  $n_{msort} = 2$ .

So, it suffices to approximate the function  $len_f(\bar{x})$ . To this aim we have adapted Podelski and Rybalchenko's method [19]. It is complete for linear ranking functions of loops of the form **while**  $B$  **do**  $S$ , where  $B(\bar{x})$  is a conjunction of linear constraints over the variables  $\bar{x}$  involved in the loop, and  $S(\bar{x}, \bar{x}')$  is a transition relation expressed as a conjunction of linear constraints over the variable values respectively before and after executing the loop body. Using these constraints over  $\bar{x}$  and  $\bar{x}'$ , the method creates another set of constraints over  $\overline{\lambda}_1$  and  $\overline{\lambda}_2$ , two lists of  $m$  non-negative variables, being  $m$  the number of restrictions contained in the conjunction of  $B(\bar{x})$  and  $S(\bar{x}, \bar{x}')$ . They prove that this set is satisfiable if and only if a linear ranking function exists for the **while**, and the ranking function itself can be synthesised from the values of  $\overline{\lambda}_1$  and  $\overline{\lambda}_2$ . More precisely, the method synthesises a vector  $\bar{r}$  of real numbers and two real constants  $\delta > 0$  and  $\delta_0$  such that:

$$\begin{cases} \bar{r} \cdot \bar{x} \geq \delta_0 & \forall \bar{x} . B(\bar{x}) \\ \bar{r} \cdot \bar{x}' \leq \bar{r} \cdot \bar{x} - \delta & \forall \bar{x}, \bar{x}' . B(\bar{x}) \wedge S(\bar{x}, \bar{x}') \end{cases}$$

These conditions — the ranking function is bounded from below and it decreases at each iteration — guarantee the termination of the loop.

The aim of [19] is proving termination and to exhibit a certificate of the proof. In this respect, *any* ranking function is a valid certificate, i.e. any valid value of the vectors  $\overline{\lambda}_1$  and  $\overline{\lambda}_2$  will suffice. Our aim is slightly different: we seek for the *least* upper bound to the length of the worst case call chain to a recursive *Core-Safe* function  $f^S$ . Then, we introduce two variations to [19]:

1. We replace the restriction  $\delta > 0$  by  $\delta \geq 1$ . In this way, each transition counts at least as an internal call to  $f^S$  and so we will get an upper bound to the number of internal calls in the chain (this would not be true if  $\delta$  were any real number satisfying  $0 < \delta < 1$ ).
2. We reformulate the problem as a *minimisation* one. We ask for the solution giving the minimum value of the following objective function:

$$Obj \stackrel{\text{def}}{=} \sum \overline{\lambda}_1 + \sum \overline{\lambda}_2 - \delta_0$$

Minimising  $-\delta_0$  is equivalent to maximising  $\delta_0$ , expressing that we look for the minimum value of  $\bar{r}.\bar{x}$  that is still an upper bound. Minimising the values of  $\bar{\lambda}_1$  and  $\bar{\lambda}_2$  is needed because we have seen that requiring only the first condition frequently leads to unbounded linear problems with an infinite number of solutions, being the minimum one the one assigning infinite to some of the components of  $\lambda_1$  or  $\lambda_2$ .

The only remaining task is codifying our abstract size functions as **while** loops. In this respect, the meaningful information is the size change between the arguments of an external call to  $f^S$  and the arguments of its internal calls. The result sizes are not relevant for termination of the call chains. But we still must decide what to do when there are more than one internal call, either excluding each other (as in  $merge^S$ ), or executed in sequence (as in  $msort^S$ ). Our approach has been in both cases to compute the convex hull of the restrictions coming from the internal calls, and to use this polyhedron both as the guard  $B(\bar{x})$  —by collecting all restrictions depending only on  $\bar{x}$ —, and as the transition relation  $S(\bar{x}, \bar{x}')$  —by collecting all restrictions depending both on  $\bar{x}$  and  $\bar{x}'$ .

- The justification of this decision in the case of excluding calls is clear: at each ‘iteration’ the function may decide to take a possible different branch, so the convex hull amounts to computing the logical ‘or’ of the restrictions coming from all the branches. In geometric terms, the ‘or’ is the convex hull.
- In the case of consecutive calls, the reasoning is different: at each internal node of the call tree, the function will take all the children branches and we seek for a bound to the worst case path. We need to collect in this case the *minimum* set of restrictions applicable to all the branches. This is also the convex hull. It is like having a loop going from the root of the tree to any leaf, which non-deterministically decides at each iteration the branch it will follow. A bound to the iterations of this ‘loop’ is then a bound to the longest path in the call tree.

## Examples

In Fig. 9 we show the restrictions of each internal call for  $split^S$ ,  $merge^S$ , and  $msort^S$ , and their respective convex hulls. When introducing this data to the above formulation of Podelski and Rybalchenkos’s method, we got the following ranking functions:

Function	$\bar{r}$	$\delta_0$	$len_f(\bar{x})$	$B(\bar{x})$
$split\ n\ x$	$[0, 1]$	2	$x$	$n \geq 1 \wedge x \geq 2$
$merge\ x\ y$	$[1, 1]$	4	$x + y - 2$	$x \geq 2 \wedge y \geq 2$
$msort\ x$	$[2]$	2	$2x$	$x \geq 3$

We have taken as  $len_f(\bar{x})$  the expression  $\bar{r}.\bar{x} - \delta_0 + 2$ , because  $\bar{r}.\bar{x} - \delta_0 + 1$  is a bound to the number of ‘iterations’, each one corresponding to an internal call to  $f^S$ , and we add 1 for taking into account the code before the loop, representing the initial call of the chain. Of course, this length is valid when  $B(\bar{x})$  holds at

Function	Internal call 1	Internal call 2	Convex hull
<i>split</i> $n\ x$	$\{n \geq 1, x \geq 2, n' = n - 1, x' = x - 1\}$		$\{n \geq 1, x \geq 2, n' = n - 1, x' = x - 1\}$
<i>merge</i> $x\ y$	$\{x \geq 2, y \geq 2, x' = x - 1, y' = y\}$	$\{x \geq 2, y \geq 2, x' = x, y' = y - 1\}$	$\{x \geq 2, y \geq 2, x + y = x' + y' + 1\}$
<i>msort</i> $x$	$\{x \geq 3, x' = \frac{1}{2}x\}$	$\{x \geq 3, x' = \frac{1}{2}x + 1\}$	$\{x \geq 3, x' \geq \frac{1}{2}x, x' \leq \frac{1}{2}x + 1\}$

**Fig. 9.** Termination restrictions of *split*, *merge* and *msort*

the beginning. Otherwise, the length is just 1. Notice that the bounds for *split* and *merge* are tight, while the one for *msort* is not (a tight bound would be  $\log_2 x + 1$ ). An obvious limitation of the method is that it can only give a linear function as a result. On the other hand, it is not so easy to find in the literature recursive functions whose longest call chain is described by a more-than-linear function.

## 6 Conclusions

We have shown that a bunch of linear techniques can be successfully applied to a first-order functional language in order to infer size invariants between the arguments and results of recursive functions, and upper bounds to the longest call chain of these functions. To this respect, some prior transformations of the program may be needed in order to distinguish between internal recursive calls related by ‘or’ (i.e. excluding each other), from those related by ‘and’ (i.e. executed in a sequence). This distinction comes for free in Prolog programs, but not in functional ones.

Linear techniques — namely abstract interpretation on polyhedral domains and linear ranking function synthesis — have been extensively used in imperative and logic languages (see e.g. [4] for a broad bibliography), but apparently there have been no much effort in applying them to functional languages. An exception regarding termination analysis (although not necessarily a linear one) is Sereni and Jones work [21] applying the SCT criterion to the termination of higher-order ML programs.

The algorithms presented here have been implemented for the moment in SWI-Prolog<sup>1</sup>, by using its CLP(Q) and simplex libraries. We have adapted to this Prolog system Andy King’s algorithm [8] for computing convex hulls. Our near future plans are to include these algorithms in the *Safe* compiler<sup>2</sup>, by using an appropriate polyhedra library.

<sup>1</sup> Available at <http://www.swi-prolog.org/>

<sup>2</sup> A web version of the compiler is available at <http://dalila.sip.ucm.es/~safe>



## References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008)
2. Alias, C., Darté, A., Feautrier, P., Gonnord, L.: Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010)
3. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: The Parma Polyhedra Library, User's Manual. Dept. of Mathematics. Univ. of Parma, Italy (2002), <http://www.cs.unipr.it/ppl/>
4. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. *Science of Computer Programming* 58(1-2), 28–56 (2005) Special Issue on the Static Analysis Symposium 2003 - SAS 2003
5. Ben-Amram, A.M., Codish, M.: A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 218–232. Springer, Heidelberg (2008)
6. Ben-Amram, A.M., Lee, C.S.: Program termination analysis in polynomial time. *ACM Transactions on Programming Languages and Systems* 29(1) (2007)
7. Benoy, F., King, A.: Inferring Argument Size Relationships with CLP(R). In: LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)
8. Benoy, F., King, A., Mesnard, F.: Computing convex hulls with a linear solver. *TPLP* 5(1-2), 259–271 (2005)
9. Colón, M., Sipma, H.: Practical Methods for Proving Program Termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL, pp. 238–252. ACM, New York (1977)
11. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL, pp. 84–96 (1978)
12. Gonnord, L., Halbwachs, N.: Combining Widening and Acceleration in Linear Relation Analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
13. Horspool, R.N.: Analyzing List Usage in Prolog Code, University of Victoria (March 1990)
14. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92 (2001)
15. Lucas, S., Peña, R.: Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In: Proc. Logic-Based Program Synthesis and Transformation, LOPSTR 2008, pp. 43–57 (2008)
16. Montenegro, M., Peña, R., Segura, C.: A Type System for Safe Memory Management and its Proof of Correctness. In: ACM Principles and Practice of Declarative Programming, PPDP 2008, Valencia, Spain, pp. 152–162 (July 2008)
17. Montenegro, M., Peña, R., Segura, C.: A Simple Region Inference Algorithm for a First-Order Functional Language. In: Escobar, S. (ed.) WFLP 2009. LNCS, vol. 5979, pp. 145–161. Springer, Heidelberg (2010)
18. Montenegro, M., Peña, R., Segura, C.: A space consumption analysis by abstract interpretation. In: van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2009. LNCS, vol. 6324, pp. 34–50. Springer, Heidelberg (2010)

19. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
20. Podelski, A., Rybalchenko, A.: Transition Invariants. In: LICS, pp. 32–41. IEEE Computer Society, Los Alamitos (2004)
21. Sereni, D., Jones, N.D.: Termination analysis of higher-order functional programs. In: Yi, K. (ed.) Programming Languages and Systems. LNCS, vol. 3780, pp. 281–297. Springer, Heidelberg (2005)

# Memoizing a Monadic Mixin DSL

Pieter Wuille<sup>1</sup>, Tom Schrijvers<sup>2</sup>, Horst Samulowitz<sup>3</sup>,  
Guido Tack<sup>1</sup>, and Peter Stuckey<sup>4</sup>

<sup>1</sup> Department of Computer Science, K.U.Leuven, Belgium

<sup>2</sup> Department of Applied Mathematics and Computer Science, UGent, Belgium

<sup>3</sup> IBM Research, USA

<sup>4</sup> National ICT Australia (NICTA) and University of Melbourne, Victoria, Australia

**Abstract.** Modular extensibility is a highly desirable property of a domain-specific language (DSL): the ability to add new features without affecting the implementation of existing features. Functional mixins (also known as open recursion) are very suitable for this purpose.

We study the use of mixins in Haskell for a modular DSL for search heuristics used in systematic solvers for combinatorial problems, that generate optimized C++ code from a high-level specification. We show how to apply memoization techniques to tackle performance issues and code explosion due to the high recursion inherent to the semantics of combinatorial search.

As such heuristics are conventionally implemented as highly entangled imperative algorithms, our Haskell mixins are monadic. Memoization of monadic components causes further complications for us to deal with.

## 1 Application Domain

Search heuristics often make all the difference between effectively solving a combinatorial problem and utter failure. Heuristics enable a search algorithm to become efficient for a variety of reasons, e.g., incorporation of domain knowledge, or randomization to avoid heavy tailed runtimes. Hence, the ability to swiftly design search heuristics that are tailored towards a problem domain is essential to performance improvement. In other words, this calls for a high-level domain-specific language (DSL).

The tough technical challenge we face when designing a DSL for search heuristics, does not lie in designing a high-level syntax; several proposals have already been made (e.g., [10]). What is really problematic is to bridge the gap between a conceptually simple specification language (high-level and naturally compositional) and an efficient implementation (typically low-level, imperative and highly non-modular). This is indeed where existing approaches fail; they restrict the expressiveness of their DSL to face up to implementation limitations, or they raise errors when the user strays out of the implemented subset.

We overcome this challenge with a systematic approach that disentangles different primitive concepts into separate modular *mixin* components, each of which corresponds to a feature in the high-level DSL. The great advantage of

```

s ::= prune
    prunes the node
  | base_search(...)
    label
  | let(v, e, s)
    introduce new global variable v with initial
    value e, then perform s
  | assign(v, e)
    assign e to variable v and succeed
  | and([s1, s2, ..., sn])
    perform s1, on success start s2 otherwise fail, ...
  | or([s1, s2, ..., sn])
    perform s1, on termination start s2, ...
  | post(c, s)
    perform s and post a constraint c at every node

```

**Fig. 1.** Syntax of Search Heuristics DSL

mixin components to provide a semantics for our DSL is its modular extensibility. We can add new features to the language by adding more mixin components. The cost of adding such a new component is small, because it does not require changes to the existing ones.

The application under consideration is heuristics for systematic tree search in the area of Constraint Programming (CP), but the same issues apply to other search-driven areas in the field of Artificial Intelligence (AI) and related areas such as Operations Research (OR). The goal is generating tight C++ code for doing search from our high-level DSL. The focus however lies in the combination of using Haskell combinators for expressing strategies, open recursion to allow modular extension and monads for allowing stateful behaviour to implement a code-generation system. Further on, we explain how to combine this with memoization to improve generation time as well as size of the generated code.

## 2 Brief DSL Overview

We provide the user with a high-level domain-specific language (DSL) for expressing search heuristics. For this DSL we use a concrete syntax, in the form of nested terms, that is compatible with the *annotation* language of MiniZinc [9], a popular language for modeling combinatorial problems.

The search specification implicitly defines a search tree whose leaves are solutions to the given problem. Our implementation parses a MiniZinc model, extracts the search specification expressed in our DSL and generates the corresponding low-level C++ code for navigating the search tree. The remainder of the MiniZinc model (expressing the actual combinatorial problem) is shipped to the Gecode library [7], a state-of-the-art finite domain constraint solver.

The search code interacts with the solver at every node of the search tree to determine whether a solution or dead end has been reached, or whether to generate new child nodes for further exploration.

## 2.1 DSL Syntax

The DSL’s *expression language* comprises the typical arithmetic and comparison operators and literals that require no further explanation. Notable though is the fact that it allows referring to the constraint variables and parameters of the constraint model.

The DSL’s *search heuristics language* features a number of primitives, listed in the catalog of Fig. 1, in terms of which more complex heuristics can be defined. The catalog consists of both *basic* heuristics and *combinators*. The former define complete (albeit very basic) heuristics by themselves, while the latter alter the behavior of one or more other heuristics.

There are two basic heuristics: **prune**, which cuts the search tree below the current node, and the base search strategies, which implement the *labeling* (also known as *enumeration*) strategies. We do not elaborate on the base search here, because this has been studied extensively in the literature. While only a few basic heuristics exist, the DSL derives great expressive power from the infinite number of ways in which these basic heuristics can be composed by means of combinators.

The combinator **let**( $v, e, s$ ) introduces a new variable  $v$ , initialized to the value of expression  $e$ , in the sub-search  $s$ , while **assign**( $v, e$ ) assigns the value of  $e$  to  $v$  and succeeds. The and-sequential composition **and**( $[s_1, \dots, s_n]$ ) runs  $s_1$  and at every success leaf runs **and**( $[s_2, \dots, s_n]$ ). In contrast, **or**( $[s_1, \dots, s_n]$ ) first runs  $s_1$  in full before restarting with **or**( $[s_2, \dots, s_n]$ ).

Finally, the **post**( $c, s$ ) primitive provides access to the underlying constraint solver, posting a constraint  $c$  at every node during  $s$ . If  $s$  is omitted, it posts the constraint and immediately succeeds.

As an example, this is how branch-and-bound — a typical optimization heuristic — can be expressed in the DSL:

```
let(best, maxint, post(obj < best, and([base_search(...), assign(best, obj)])))
```

**let** introduces the variable *best*, **post** makes sure the constraint  $obj < best$  is enforced at each node of the search tree spawned by **base\_search**. Combining it with **assign** using **and** causes the *best* variable to be updated after finding solutions. Note that we refer to *obj*, the program variable being minimized.

## 3 Implementation

Starting from base searches and functions for combining them — as called by the parser — a C++ AST is generated. After a simplification step, a pretty printer is invoked to generate the actual source code. Both the initial parsing phase and pretty printer are trivial and not discussed here.

### 3.1 C++ Abstract Syntax Tree

Before we discuss the code generator, we need to define the target language, a C++ AST, which is partly given here:

```

data Stmt = Nop                               | Expr := Expr
          | IfThenElse Expr Stmt Stmt          | Stmt ; Stmt
          | Call String [Expr]                 | While Expr Stmt
          | ...

```

A number of convenient abbreviations facilitate building this AST, e.g.,

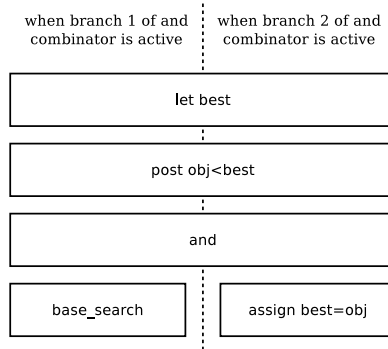
```

(;) = liftM  o (;)
if' = liftM2 o IfThenElse

```

### 3.2 The Combinator Stack

Based on the output of the parser, a data structure is built that represents the search heuristic. The details of how this is represented will follow later, but in general, a value of type *Search* will be used. Basic heuristics result immediately in a *Search*, while combinators are modeled as functions that take one or more *Search* values, and compute a derived one from that. Although conceptually this is best modeled as a tree structure, with each subtree evaluating to a *Search*, processing happens top-down, and only a single path through the combinator tree is active at a given time. The list of combinators along this path will be called the combinator stack. Figure 2 shows the combinator stack for the earlier branch-and-bound example.



**Fig. 2.** Branch-and-bound combinator stack

### 3.3 The Code Generator

Inside *Search* structures, values of type *Gen m* will be built up. They contain a number of hooks that produce the corresponding AST fragments<sup>1</sup>.

As will be explained later, some combinators need to keep an own modifiable state during code generation, so hooks must support side effects; hence *Gen* is parametrized in a monad *m*.

```
data Gen m = Gen { initG  :: m Stmt, bodyG :: m Stmt
                  , addG  :: m Stmt, tryG  :: m Stmt
                  , resultG :: m Stmt, failG :: m Stmt
                  , height :: Int }
```

The separate hooks correspond to several stages for the processing of nodes in a search tree. Nodes are initialized with *init<sub>G</sub>* and processed using consecutively *body<sub>G</sub>*, *add<sub>G</sub>*, and *try<sub>G</sub>*. *result<sub>G</sub>* is used for reporting solutions, and *fail<sub>G</sub>* for aborting after failure. The *height* field indicates how high the stack of combinators is.

The fragments of the different hooks are combined according to the following template.

```
gen :: Monad m => Gen m -> m Stmt
gen g = do init ← initG g
        try  ← tryG g
        body ← bodyG g
        return $ declarations
                ; init
                ; try
                ; While queueNotEmpty body
```

After emitting a number of variable declarations which we omit due to space constraints, the template creates the root node in the search tree through *init<sub>G</sub>*, and *try<sub>G</sub>* initializes a queue with child nodes of the root. Then, in the main part of the algorithm, nodes in the queue are processed one at a time with the *body<sub>G</sub>* hook.

### 3.4 Code Generation Mixins

Instead of writing a monolithic code generator for every different search heuristic, we modularly compose new heuristics from one or more components, each of which corresponds to a constructor in the high-level DSL. Our code generator components are implemented as (functional) mixins [2], where the result is a function from *Eval m* to *Eval m*, which gets called with its own resulting strategy as argument. The function argument in these mixins is comparable to the *this* object in object-oriented paradigms.

<sup>1</sup> See Section 3.4 for why we partition the code generation into these hooks.





```

Gen { initG   = return Nop
      , bodyG  = addG this
      , addG   = constrain ; tryG this
      , tryG   = let ret  = resultG this
                  succ = if' isSolved ret doBranch
                  in if' isFailed (failG this) succ
      , resultG = return Nop
      , failG  = return Nop
      , height  = 0 }

```

The above code omits details related to posting constraints (*constrain*), checking the solver status (*isSolved* or *isFailed*) and branching (*doBranch*). The details of these operations depend on the particular constraint solver involved (e.g. finite domain, linear programming, ...); here we focus only on the search heuristics, which are orthogonal to those details.

As we can see the base component is parametrized by *this*, the overall search heuristic. This way, the *base<sub>M</sub>* search can make the final call to *body<sub>G</sub>* redirect to an *add<sub>G</sub>* on the top of the combinator-stack again, restarting the processing top-down, but this time using *add<sub>G</sub>* instead of *body<sub>G</sub>*. A similar construct is used for called *try<sub>G</sub>* and *result<sub>G</sub>*.

The simplest form of a search heuristic is obtained by applying the fix-point combinator to a base component:

```

fix :: Mixin a → a
fix m = m (fix m)
search1 :: Gen Identity
search1 = fix baseM

```

*Advice Component.* The mixin mechanism allows us to plug in additional advice components before applying the fix-point combinator. This way we can modify the base component's behavior.

Consider a simple example of an advice combinator that prints solutions:

```

printM :: Monad m ⇒ MGen m
printM super = super { resultG = printSolution ; resultG super
                      , height  = 1 + height super }

```

where *printSolution* consists of the necessary solver-specific code to access and print the solution. A code generator is obtained through mixin composition, simply using (*o*):

```

search2 :: Gen Identity
search2 = fix (printM o baseM)

```

### 3.5 Monadic Components

In the components we have seen so far, the monad type parameter *m* has not been used. It does become essential when we turn to more complex components such as the binary conjunction *and*([*g<sub>1</sub>*, *g<sub>2</sub>*]).

The code presented at the end of this section shows a simplified *and* combinator, for two *Gen m* structures with the same type *m*. It does require *m* to be an instance of *MonadReader Side*, to store the current branch at code-generation runtime. While some hooks simply dispatch to the corresponding hook of the currently active branch, *body<sub>G</sub>* and *result<sub>G</sub>* are more elaborate.

First of all, we also need to store the branch number at program runtime. This is known at the time when the node is created, but needs to be restored into the monadic state when activating it. We assume the functions *store* and *retrieve* give access to a runtime state for each node, indexed with a field name and the height of the combinator involved.

When the *result<sub>G</sub>* hook is called — implying a solution for a sub-branch was found — there are two options. Either the *g<sub>1</sub>* was active, in which case both the runtime state and the monadic state are updated to *In<sub>2</sub>*, and *init<sub>G</sub>* and *try<sub>G</sub>* for *g<sub>2</sub>* are executed, which will possibly cause the node to be added to the queue, if branching is required. When this new node is activated itself, its *body<sub>G</sub>* hook will be called, retrieving the branch information from the runtime state, and dispatching dynamically to *g<sub>2</sub>*. When a solution is reached after switching to *g<sub>2</sub>*, *result<sub>G</sub>* will finally call *g<sub>2</sub>*'s *result<sub>G</sub>* to report the full solution.

```

data Branch = In1 | In2
type Mixin2 a = a → a → a
andM :: MonadReader Branch m ⇒ Mixin2 (Gen m)
andM g1 g2 = Gen { initG   = store myHeight "pos" In1 ; initG g1
                      , addG   = dispatch addG
                      , tryG   = dispatch tryG
                      , failG  = dispatch failG
                      , bodyG  = myBody
                      , resultG = myResult
                      , height  = myHeight }
where parent      = ask >>= λx → case x of
                      In1 → return g1
                      In2 → return g2
dispatch f = parent >>= f
myHeight  = 1 + max (height g1) (height g2)
myBody    = let pos = retrieve myHeight "pos"
           br1 = local (const In1) (bodyG g1)
           br2 = local (const In2) (bodyG g2)
           in if' (pos := In1) br1 br2
myResult  = do num ← ask
           case num of
             In1 → local (const In2) $
                   store myHeight "pos" In2
                   ; liftM2 (:) (initG g2) (tryG g2)
             In2 → resultG g2

```

### 3.6 Effect Encapsulation

So far we have parametrized  $MGen$  with  $m$ , a monad type parameter. This parameter will have to be assembled appropriately from monad transformers to satisfy the need of every mixin component in the code generator. Doing this manually can be quite cumbersome. Especially for a large number of mixin components with multiple instances of, e.g., *StateT* this becomes impractical. To simplify the process, we turn to a technique proposed by Schrijvers and Oliveira [11] to encapsulate the monad transformers inside the components.

**data**  $Search = \forall t_2. MonadTrans\ t_2 \Rightarrow$   
 $Search\ \{mgen :: \forall m\ t_1. (Monad\ m, MonadTrans\ t_1) \Rightarrow MGen\ ((t_1 \triangleright t_2)\ m)$   
 $, run :: \forall m\ x. Monad\ m \Rightarrow t_2\ m\ x \rightarrow m\ x\}$

To that end we now represent components by the *Search* type that was announced earlier, which packages the components behavior  $MGen$  with its side effect  $t_2$ . The monad transformer  $t_2$  is existentially quantified to remain hidden; we can eliminate it from a monad stack with the *run* field. The hooks of the component are available through the *mgen* field, which specifies them for an arbitrary monad stack in which  $t_2$  is surrounded by more effects  $t_1$  above and  $m$  below. Here  $t_1 \triangleright t_2$  indicates that the focus rests on  $t_2$  (away from  $t_1$ ) for resolving overloaded monadic primitives such as *get* and *put*, for which multiple implementations may be available in the monad stack. We refer to [12,11] for details of this focusing mechanism, known as the *monad zipper*.

An auxiliary function promotes a non-effectful  $MGen\ m$  to  $MSearch$ :

**type**  $MSearch = Mixin\ Search$   
 $mkSearch :: (\forall m. Monad\ m \Rightarrow MGen\ m) \rightarrow MSearch$   
 $mkSearch\ f\ super =$   
**case** *super* **of**  
 $Search\ \{mgen = mgen, run = run\} \rightarrow Search\ \{mgen = f \circ mgen$   
 $, run = run\}$

which we can apply for instance to  $base_M$  and  $print_M$ .

$base_S, print_S :: MSearch$   
 $base_S = mkSearch\ base_M$   
 $print_S = mkSearch\ print_M$

Similarly, we define  $mkSearch_2$  for lifting binary combinators like  $and_M$ . It takes a combinator for two  $Gen\ m$ 's, as well as a run function for additional monad transformers the combinator may require, and lifts it to  $MSearch_2$  (implementation omitted).

**type**  $MSearch_2 = Mixin_2\ Search$   
 $and_S :: MSearch_2$   
 $and_S = mkSearch_2\ and_M\ (flip\ runReaderT\ In_1)$

$$\begin{aligned}
mkSearch_2 &:: MonadTrans\ t_2 \\
&\Rightarrow (\forall m\ t_1. (Monad\ m, MonadTrans\ t_1) \Rightarrow Mixin_2\ (Gen\ ((t_1 \triangleright t_2)\ m))) \\
&\rightarrow (\forall m\ x. Monad\ m \Rightarrow t_2\ m\ x \rightarrow m\ x) \\
&\rightarrow MSearch_2
\end{aligned}$$

Finally we produce C++ code from a *Search* component with *generate*:

```

generate :: Search → Stmt
generate s = case s of
    Search { mgen = mgen, run = run } →
        runIdentity $ run $ runIdentityT $ runZ $ gen $ fix $ mgen

```

This code first applies the fix-point computation, passing the result back into itself, as explained earlier. After that, *gen* is called to get the real code-generating monad action. It extracts the knot-tied *body<sub>G</sub>* hook, *runZ* eliminates  $\triangleright$  from  $(t_1 \triangleright t_2)\ m$ , yielding  $t_1\ (t_2\ m)$ . Then *runIdentityT* eliminates  $t_1$  (instantiating it to be *IdentityT*), *run* eliminates  $t_2$ , and *runIdentity* finally eliminates  $m$  (instantiating it to be *Identity*) to yield a *Stmt*.

## 4 Memoization and Inlining

Experimental evaluation indicates that several component hooks in a complex search heuristic are called frequently, as for example the *fail<sub>G</sub>* hook can be called from many different places. This is a problem 1) for the code generation — which needs to generate the corresponding code over and over again — and 2) for the generated program which contains much redundant code. Both significantly impact the compilation time (in Haskell and in C++); in addition, an overly large binary executable may adversely affect the cache and ultimately the running time.

### 4.1 Basic Memoization

A well-known approach that avoids the first problem, repeatedly computing the same result, is *memoization*. Fortunately, Brown and Cook [4] have shown that memoization can be added as a monadic mixin component without any major complications.

Memoization is a side effect for which we define a custom monad transformer:

```

newtype MT m a = MT { runMT :: StateT Table m a }
deriving (MonadTrans)

runMemoT :: Monad m ⇒ MT m a → m (a, Table)
runMemoT m = runStateT (runMT m) initMemoState

```

which is essentially a state transformer that maintains a table from *Keys* to *Stmts*. For now we use *Strings* as *Keys*.

```

newtype Key   = String
newtype Table = Map Key Stmt
initMemoState = empty

```

We capture the two essential operations of  $\mathbb{M}_T$  in a type class, which allows us to lift the operations through other monad transformers<sup>2</sup>

```

class Monad m  $\Rightarrow$   $\mathbb{M}_M$  m where
  getM :: String  $\rightarrow$  m (Maybe Stmt)
  putM :: String  $\rightarrow$  Stmt  $\rightarrow$  m ()
instance Monad m  $\Rightarrow$   $\mathbb{M}_M$  ( $\mathbb{M}_T$  m) where ...
instance ( $\mathbb{M}_M$  m, MonadTrans t)  $\Rightarrow$   $\mathbb{M}_M$  (t m) where ...

```

These operations are used in an auxiliary mixin function:

```

memo ::  $\mathbb{M}_M$  m  $\Rightarrow$  String  $\rightarrow$  Mixin (m Stmt)
memo s m = do stm  $\leftarrow$  getM s
           case stm of
             Nothing  $\rightarrow$  do code  $\leftarrow$  m
                           putM s code
                           return code
             Just code  $\rightarrow$  return code

```

which is used by the advice component:

```

memo_M ::  $\mathbb{M}_M$  m  $\Rightarrow$  MGen m
memo_M super = super { init_G = memo "init" (init_G super)
                      , body_G = memo "body" (body_G super)
                      , add_G = memo "add" (add_G super)
                      , try_G = memo "try" (try_G super)
                      , result_G = memo "result" (result_G super)
                      , fail_G = memo "fail" (fail_G super) }

```

which allows us to define, e.g., a memoized variant of  $print_S$ .

```

print_S = mkSearch (memo_M  $\circ$  print_M)

```

Note that in order to lift  $memo_M$  to a *Search* structure, *Search* must be updated with a  $\mathbb{M}_M$  m constraint, and *generate* must be updated to incorporate *runMemoT* in its evaluation chain.

```

data Search =  $\forall t_2. \text{MonadTrans } t_2 \Rightarrow$ 
  Search { mgen ::  $\forall m. t_1. (\mathbb{M}_M m, \text{MonadTrans } t_1) \Rightarrow \text{MGen } ((t_1 \triangleright t_2) m)$ 
        , run   ::  $\forall m. x. \mathbb{M}_M m \Rightarrow t_2 m x \rightarrow m x$  }
generate s =
  case s of
    Search { mgen = mgen, run = run }  $\rightarrow$ 
      runIdentity $ runMemoT $ run $ runIdentityT $ runZ $ gen $ fix mgen

```

---

<sup>2</sup> For lack of space we omit the straightforward instance implementations.

## 4.2 Monadic Memoization

Unfortunately, it is not quite this simple. The behavior of combinator hooks may depend on internal updateable state, like  $and_M$  from section 3.5 kept a *Branch* value as state. The above memoization does not take this state dependency into account.

In order to solve this issue, we must expose the components' state to the memoizer. This is done in two steps. First,  $\mathbb{M}_T$  keeps a *context* in addition to the memoization table, and provides access to it through the  $\mathbb{M}_M$  type class. Second — for the specific case of a *ReaderT s* with *s* an instance of *Showable* — an alternative implementation (*MemoReaderT*) which updates the context in the  $\mathbb{M}_T$  layer below it, is provided. Typically, the used states are simple in structure.

To implement this, the *Table* type is extended:

```
type MemoContext = Map Int String
type Key          = (MemoContext, String)
data Table = Table { context :: MemoContext
                    , memoMap :: Map Key Stmt }
initMemoState = Table { context    = empty
                      , memoMap = empty }
```

*MemoContext* is represented as a map from integers to strings. The integers are identifiers assigned to the monad transformer layers that have context, and the strings are serialized versions of the contextual data inside those layers (using *show*).

The  $\mathbb{M}_M$  type class is extended to support modifying the context information, using *setCtx* and *clearCtx*.

```
class Monad m  $\Rightarrow$   $\mathbb{M}_M$  m where
  ...
  setCtx :: Int  $\rightarrow$  String  $\rightarrow$  m ()
  clearCtx :: Int  $\rightarrow$  m ()
```

Finally,  $\mathbb{M}R_T$  is introduced. It will contain a wrapped double *ReaderT*-transformed monad. The state will be stored in the first, while the second is used to give access to the identifier of the layer.

```
newtype  $\mathbb{M}R_T$  s m a =  $\mathbb{M}R_T$  { run $\mathbb{M}R_T$  :: ReaderT Int (ReaderT s m) a }
```

For convenience,  $\mathbb{M}R_T$  is made an instance of *MonadReader*, so switching from *ReaderT* to  $\mathbb{M}R_T$  does not require any changes to the code interacting with it.

When running a  $\mathbb{M}R_T$  transformer, the enclosing *Gen*'s *height* parameter is passed to *rReaderT*, using that as identifier for the layer. The runtime state itself is stored inside the wrapped *ReaderT* layer, while a serialized representation (using *show*) is stored in the context of the underlying  $\mathbb{M}_T$ . Note that *show* implementations are supposed to turn a value into equivalent Haskell source code

for reconstructing the value — this is far from the most efficient solution, but it does produce canonical descriptions for all values, and default implementations are provided by the system for almost all useful data types. There are alternatives, such as using an *Ord*-providing *Dynamic*-like type, but those are harder to implement and there is little to be gained, as will be shown in the evaluation (Section 5).

```

instance (Show s,  $\mathbb{M}_M$  m)  $\Rightarrow$  MonadReader s (MIRT s m) where
  ask = MIRT $ lift ask
  local s m = MIRT $ do n  $\leftarrow$  ask
                        old  $\leftarrow$  lift ask
                        let new = s old
                        putCtx n $ show new
                        let im = runMIRT m
                        r  $\leftarrow$  mapReaderT (local $ const new) im
                        putCtx n $ show old
                        return r

rMIRT :: ( $\mathbb{M}_M$  m, Show s)  $\Rightarrow$  s  $\rightarrow$  Int  $\rightarrow$  MIRT s m a  $\rightarrow$  m a
rMIRT s height m =
  do let action = runReaderT (runMIRT m) height
      putCtx height (show s)
      result  $\leftarrow$  runReaderT action s
      clearCtx height
      return result

```

### 4.3 Backend Sharing

So far we have only solved the first performance problem, repeated generation of code. Memoization avoids the repeated execution of hooks by storing and reusing the same C++ code fragment. However, the second performance problem, repeated output of the same C++ code, remains.

We preserve the sharing obtained through memoization in the backend, by depositing the memoized code fragment in a C++ function that is called from multiple sites. Conceptually, this means that a memoized hook returns a function call (rather than a potentially big code fragment), and produces a function definition as a side effect<sup>3</sup>.

```

memo2 ::  $\mathbb{M}_M$  m  $\Rightarrow$  String  $\rightarrow$  Mixin (m Stmt)
memo2 s m = do code  $\leftarrow$  memo s m
            let name = getFnName code
            return (Call name [])

getFnName :: Stmt  $\rightarrow$  String

```

<sup>3</sup> The function *getFnName* — given without implementation — derives a unique function name for a given code fragment.

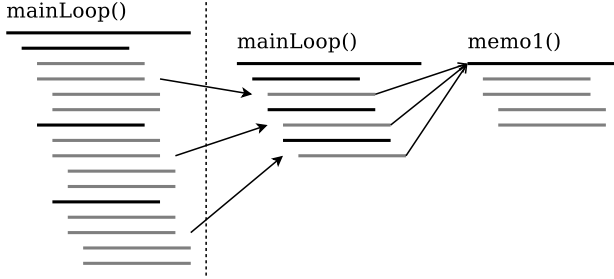
The following *generate* function produces both the main search code and the auxiliary functions for the memoized hooks. By introducing *runMemoT* in the chain of evaluation functions, the types change, and the result will be of type  $(Stmt, Table)$ , since that is returned by *runMemoT*.

```

data FunDef = FunDef String Stmt
toFunDef :: Stmt → FunDef
toFunDef stm = FunDef (getFnName stm) stm
generate :: Search → (Stmt, [FunDef])
generate s =
  case s of
    Search { mgen = mgen, run = run } →
      let eval      = fix mgen
          codeM    = gen eval
          memoM    = run ∘ runIdentityT ∘ runZ $ codeM
          (code, state) = runIdentity $ runMemoT memoM
      in (code, map toFunDef ∘ elems $ memoMap state)

```

The result of extracting common pieces of code into separate functions, is shown schematically in figure 4.



**Fig. 4.** Memoization with auxiliary functions

Note that only code generated by the same hook of the same component is shared in a function, not code of distinct hooks or distinct components. Separate from the mechanism described above, it is also possible to detect unrelated *clones* by doing memoization with only the generated code itself as key (instead of function names, present variables and active states). This causes a slowdown, as the code needs to be generated for each instance before it can be recognized as identical to earlier emitted code. To a limited extent, this second memoization scheme is also used in the implementation to reduce the size of generated code — without any measurable overhead.

Finally, applying the above technique systematically results in one generated C++ function per component hook. This is not entirely satisfactory, as many



memoized functions are only called once, or only contain a single line of code. One can either rely on the C++ compiler to determine when inlining is lucrative, or perform inlining on the C++ AST in an additional processing step.

## 5 Evaluation

We have omitted a number of complicating factors in our account, so as not to distract from the main issues. Without going into detail, we list the main differences with the actual implementation:

- There are more hooks, including ones called during branching, adding to the queue, deletion of nodes and switching between nodes belonging to separate strategies. Furthermore, additional hooks exist for the creation of combinator-specific data structures, both globally for the whole combinator, or locally for each node, instead of the dynamic *height*-based mechanism.
- The code generation hooks are functions that take an additional argument, the *path info*. It contains which variable names point to the local and global data structures, which variables need to be passed to generated memoized functions, and pieces of code that need to be executed when the current node needs to be stored, aborted or copied. The values in the path info are also taken into account when memoizing, complicating matters further.
- We have built into the code generators a number of optimizations. For example, if it is known that a combinator never branches, certain generated code and data structures may be omitted.
- Searches keep track of whether they complete exhaustively, or are pruned. Repeat-like combinators use exhaustiveness as an additional stop criterion.

To evaluate the usefulness of our system, benchmarks<sup>4</sup> were performed (see Table 1)<sup>5</sup>. A first set includes the known problems **golfers**<sup>6</sup>, **golomb**<sup>7</sup>, **open stacks** and **radiation**<sup>11</sup>; a second set contains artificial stress tests. The different problem sizes for **golomb** use the same search code, while in **ortest** and **radiation**, separate code is used.

The first three columns give the name, problem size and whether or not the memoizing version was used. Further columns show the number of generated C++ lines (col. 4), the number of invoked hooks (col. 5), the number of monad transformers active (both the effective ones (col. 6), and including *IdentityT* and  $\triangleright$  (col. 7)). Finally, the average generation (Haskell, col. 8), build (gcc, col. 9) and run time (col. 10) are listed. All these numbers are averages over many runs (of up to an hour of runtime).

<sup>4</sup> Available at <http://users.ugent.be/~tschrijv/SearchCombinators>

<sup>5</sup> A 2.13GHz Intel(R) Core(TM)2 Duo 6400 system, with 2GiB of RAM was used. The system was running Ubuntu 10.10 64-bit, with GCC 4.4.4, Gecode 3.3.1 and Minizinc 1.3.1.

<sup>6</sup> Social golfer problem, CSPlib problem 10.

<sup>7</sup> Golomb rulers, CSPlib problem 6.

**Table 1.** Benchmark results

name	size	memo?	lines	hooks	trans. eff. total		time generate build run		
golomb	10	no	216	70	4	14	0.00017	2.0	4.9
		yes	187	95	5	17	0.0073	2.0	4.9
	11	no							110
		yes							110
	12	no							1200
		yes							1200
open-stacks	30	no	216	70	4	14	0.00016	2.1	0.12
		yes	187	95	5	17	0.0074	2.0	0.12
golfers		no	119	29	3	8	0.00017	2.0	1.3
		yes	114	46	4	11	0.00017	2.0	1.3
radiation	15	no	11455	4153	4	76	0.57	16	210
		yes	2193	1155	5	79	0.19	4.0	230
	5	no	2530	898	4	36	0.073	4.3	0.10
		yes	933	485	5	39	0.055	2.7	0.10
bab-real		no	216	70	4	14	0.00019	2.0	17
		yes	187	95	5	17	0.0074	2.0	17
bab-restart		no	1499	1166	5	20	0.045	2.8	17
		yes	433	262	6	23	0.026	2.2	17
for+copy		no	1164	414	5	14	0.016	2.4	8.9
		yes	494	180	6	17	0.0066	2.1	8.9
once-sequence		no	2530	898	4	36	0.073	4.2	2.7
		yes	933	485	5	39	0.054	2.7	2.6
ortest	10	no	1597	849	13	48	0.11	3.2	17
		yes	1222	655	14	51	0.11	2.6	17
	20	no	4232	1869	23	88	0.82	9.7	17
		yes	3352	1465	24	91	0.79	6.7	17

For the larger problem instances, memoization reduces both generation time and build time, by reducing the number of generated lines. No reduced cache effects resulting from memoizing large generated code are observed in these examples, but performance is not affected either by the increased number of function calls. In particular for the `radiation` example, the effect of memoization is drastic. On the other hand, for small problems, memoization does not help, but the overhead is very small.

## 6 Related Work

We were inspired by the monadic mixin approach to memoization of Brown and Cook [4]. The problem of memoization of stateful monad components is not yet solved in general, but typically requires some way for exposing the implicit state, as shown in [3] for parser combinators. In our system, this is accomplished by also memoizing the implicit state.

A different approach that results in smaller code generated from a DSL is *observable sharing* [5,8]. Yet, the main intent of observable sharing is quite different. Its aim is to preserve sharing at the level of Haskell in the resulting generated code, typically using *unsafePerformIO*. It does not detect distinct calls that result in the same code, and is hard to integrate with code-generating monadic computations as appear in our setting.

Our work is directly inspired by earlier work on the Monadic Constraint Programming DSL [13,15]. In particular, we have studied how to compile high-level problem specifications in Haskell to C++ code for the Gecode library [14]. The present complements this with high-level search specifications.

## 7 Conclusions

We have shown how to implement a code generator for declarative specification of a search heuristic using monadic mixins. Using this mixin-based approach, search combinators can be implemented in a modular way, and still independently modify the behavior of the generated code. Through existential types and the monad zipper, all combinators can introduce their own monad transformers to keep their own state throughout the code generation, without affecting any other transformers.

Since the naive approach leads to certain hooks being invoked many times over, we turn to memoization to avoid code duplication. Memoization is implemented as another monadic mixin which is added transparently to existing combinators.

The system is implemented as a Haskell program that generates search code in C++ from a search specification in MiniZinc which is then further integrated in a CP solver (Gecode). Our benchmarks demonstrate the impact of memoizing the monadic mixins.

## References

1. Baatar, D., Boland, N., Brand, S., Stuckey, P.: CP and IP approaches to cancer radiotherapy delivery optimization. *Constraints* (2011)
2. Bracha, G., Cook, W.R.: Mixin-based inheritance. In: *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 303–311 (1990)
3. Brown, D., Cook, W.R.: Function inheritance: Monadic memoization mixins. Report, Department of Computer Sciences, University of Texas at Austin (June 2006)
4. Brown, D., Cook, W.R.: Function inheritance: Monadic memoization mixins. In: *Brazilian Symposium on Programming Languages, SBLP* (2009)
5. Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: *Thiagarajan, P.S., Yap, R.H.C. (eds.) ASIAN 1999. LNCS, vol. 1742*, pp. 62–73. Springer, Heidelberg (1999)
6. Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: Effective advice: disciplined advice with explicit effects. In: *Jézéquel, J.-M., Südholt, M. (eds.) AOSD*, pp. 109–120. ACM, New York (2010)

7. Gecode Team. Gecode: Generic constraint development environment (2006), <http://www.gecode.org>
8. Gill, A.: Type-safe observable sharing in haskell. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, pp. 117–128. ACM, New York (2009)
9. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
10. Samulowitz, H., Tack, G., Fischer, J., Wallace, M., Stuckey, P.: Towards a lightweight standard search language. In: Pearson, J., Mancini, T. (eds.) Constraint Modeling and Reformulation, ModRef 2010 (2010)
11. Schrijvers, T., Oliveira, B.: Modular components with monadic effects. In: Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010), vol. (UU-CS-2010-020), pp. 264–277 (2010)
12. Schrijvers, T., Oliveira, B.: The monad zipper. Report CW 595, Dept. of Computer Science, K.U.Leuven (2010)
13. Schrijvers, T., Stuckey, P.J., Wadler, P.: Monadic constraint programming. *Journal of Functional Programming* 19(6), 663–697 (2009)
14. Wuille, P., Schrijvers, T.: Monadic Constraint Programming with Gecode. In: Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation, pp. 171–185 (2009)
15. Wuille, P., Schrijvers, T.: Parameterized models for on-line and off-line use. In: Mariño, J. (ed.) WFLP 2010. LNCS, vol. 6559, pp. 101–118. Springer, Heidelberg (2011)

# A Functional Approach to Worst-Case Execution Time Analysis

Vítor Rodrigues<sup>1</sup>, Mário Florido<sup>1</sup>, and Simão Melo de Sousa<sup>2</sup>

<sup>1</sup> DCC-Faculty of Science & LIACC, University of Porto  
vitor.gabriel.rodrigues@gmail.com,  
amf@ncc.up.pt

<sup>2</sup> DI-Beira Interior University & LIACC, University of Porto  
desousa@di.ubi.pt

**Abstract.** Modern hard real-time systems demand *safe* determination of bounds on the execution times of programs. To this purpose, program execution for all possible combinations of input values is impracticable. In alternative, static analysis methods provide sound and efficient mechanisms for determining execution time bounds, regardless of input data.

We present a calculation-based and compositional development of a functional static analyzer using the Abstract Interpretation framework. Meanings of programs are expressed in fixpoint form, using a two-level denotational meta-language. At the higher level, we devise a uniform fix-point semantics with a relational-algebraic shape, defined as the reflexive transitive closure of the program binary relations. Fixpoints are calculated in the point-free style using functional composition and a proper recursive operator. At the lower level, state transformations are specified by semantic transformers designed as abstract interpretations of the transition semantics.

## 1 Introduction

The design of hard real-time systems is directed by the timeliness criteria. This safety criteria [10] is most commonly specified by the worst-case execution time (WCET) of the program, i.e. the program path that takes the longest time to execute. In the general case, the particular input data that causes the actual WCET is unknown and determination of the WCET throughout testing is an expensive process that cannot be proven correct for *any* possible run of the program. The obvious alternative to this incomputable problem are the static analysis methods, which always determine sound properties about programs.

Static analysis methods exhibit a compromise between the efficiency and precision of the process, i.e. approximations are necessary to ensure that the static analyzer stabilizes after finitely many steps. In this paper, we present a generic framework for static determination of dynamic properties of programs, based on the theory of abstract interpretation [54]. A particular instantiation of the framework for the WCET analysis is described in detail.

This work belongs to the area of practical applications of Haskell: here we present a high-level declarative implementation<sup>1</sup> in Haskell of a WCET analysis at hardware level. Haskell has several advantages for this purpose: its polymorphic type system gives us a definition of a polymorphic meta-language and a parametrized fixpoint semantics for free, the compositional aspect of the analyzer is trivially implemented by functional composition and, finally, Haskell definitions, being highly declarative, are a direct implementation of the corresponding definitions from the abstract interpretation literature.

The safeness requirement of the real-time system must be balanced with the accuracy requirement to avoid excessive WCET overestimation. In this case, tight upper-bounds on the actual WCET depend on accurate program flow information, such as infeasible program paths and maximal number of iterations in loops [7]. Our framework provides a uniform fixpoint semantics[4], which allow the analysis of program flow information as an instrumented version of the static analyzer. This objective is accomplished by a polymorphic and graph-based representation of control flow graphs.

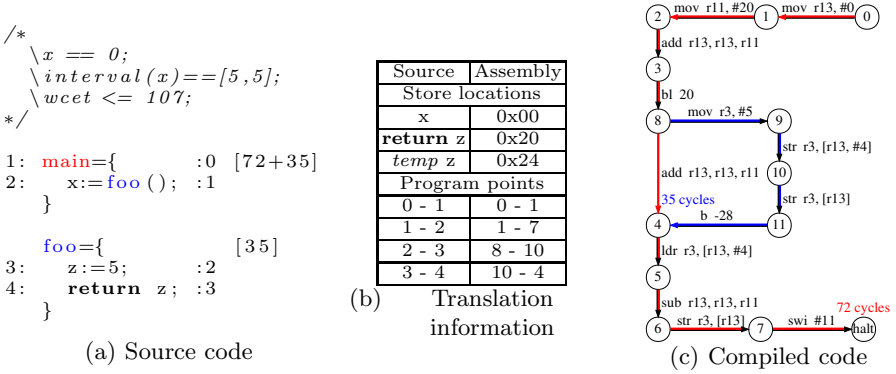
In modern real-time systems, the use of caches and pipelines considerably increases memory accesses and clock rates. Therefore, precise WCET analysis must be made at hardware level, using a timing model capable to assess the effect of each component that affect the WCET. In particular, we are interested in the computation of the execution time for every instruction in the program, taking into consideration all possible program paths. However, the WCET estimation requires a final step: the calculation step that combines program flow analysis with low-level analysis in order to find the worst-case program path [11]. To this end, a path analysis is performed *a posteriori*, using Integer Linear Programming (ILP) techniques in order to compute the WCET.

The contributions of the paper are:

- definition of a polymorphic, denotational meta-language capable to express different abstract interpretations of the same program;
- definition of a compositional and generic static analyzer which uses the meta-language to efficiently compute fixpoints using *chaotic iterations*;
- definition of a uniform fixpoint semantics capable to simultaneously integrate value analysis, cache prediction and pipeline simulation.

As an example, consider the simple program of Figure 1. The source code in Figure 1(a) includes two procedures and a timeliness safety requirement in the form of a post-condition ( $\backslash w_{cet} \leq 75$ ). The code generator produces the debug information in Figure 1(b), where a correspondence is established between source program points and assembly points. This allows the annotation of invariants computed at machine level back to the source level by means of a back-annotation mechanism[14]. Figure 1(c) shows the labeled control flow graph of the machine program, where instructions are represented by graph edges and program points are represented by graph nodes. The labeling of nodes is assigned in weak topological order[2]. The WCET of the two procedures, `main` and `foo`, is calculated

<sup>1</sup> Available in <http://www.dcc.fc.up.pt/~vitor.rodriques>



**Fig. 1.** Simple example of a WCET-aware tool chain development

and included in the control flow graph. The set of instructions associated with the first procedure takes 50 processor cycles to complete and the second procedure takes 22 processor cycles. Using the debug information in [1\(b\)](#), the WCETs are inserted in the source code as annotations. Since a call to the procedure `foo` is found inside `main`, the total WCET of the program is  $72 + 35 = 107$  processor cycles.

The remainder of this paper is organized as follows. In the next section we present the related work. In [section 3](#), we describe the two-level meta-language and in [Section 4](#) is defined our fixpoint semantics. Implementation guidelines of the static analyzer are given in [Section 5](#). [Section 6](#) concludes.

## 2 Related Work

The general applicability of abstract interpretation to programs in a wide class of languages has been assessed by Jones and Nielson in [\[9\]](#), where a denotational meta-language maps programs to the mathematical domain of typed  $\lambda$ -calculus. A distinguishable feature of our framework is the use of a relational algebra as meta-language. Relations are represented as sets of Cartesian products and operations over relations are expressed using a point-free notation. The main advantage of this approach is the algebraic aspect of fixpoint semantics: it provides a good basis for modularity in program semantics, e.g. support for interprocedural analysis [\[6\]](#), and a basis for program transformation [\[3\]](#).

Reinhard Wilhelm *et al.* has extensive research work in the application of abstract interpretation theory to WCET timing validation [\[18,15\]](#). The commercial tool **AbstInt** from AiT is the *state of the art* in respect to timing analysis. Their methods require explicit annotation of machine code to determine the number of times a loop is iterated and the static analyzer needs several runs in order to aggregate each of the value, cache and pipeline analysis results. Moreover, pipeline analysis is an hybrid static analysis since it employs both state transversal and join operations. In contrast, our framework provides a uniform fixpoint

semantics capable to incorporate the pipeline temporal evolution along the chaotic fixpoint iterations. Further, we automatically derive safe and tight annotations of infeasible paths and loops by means of an instrumented static analysis.

### 3 The Two-Level Meta-language

We propose a relational-algebraic treatment of program semantics, combined with denotational definitions of programming language semantics. The pragmatics of this separation is that we can specify the structure of programs using polymorphic relational operators and then simulate this specification by providing denotational definitions as arguments. To this end, we employ a modified version of the two-level denotational meta-language defined in [12].

The two levels of the meta-language separates compile-time entities  $ct$  from run-time entities  $rt$ . The relational algebra of run-time entities provides a point-free perspective over functional application [3]. In the point-free view, program points are not explicitly accessible in the program semantics and functions are treated as relations. The main advantage of this approach is the possibility to build new functions by combining simpler ones. In this way, we express the compile-time part of program semantics using generic relational point-free operators such as the sequential composition ( $*$ ), the parallel composition ( $/$ ), and the recursive composition ( $+$ ). On the other hand, run-time entities are viewed as “code” at compile-time and constitute the actual arguments of the relational operators. Intuitively, when interpreting the relational *meta*-program  $ct$ , variable effects can be obtained by executing different pieces of code of type  $rt_1 \rightrightarrows rt_2$ .

The compile-time entities denote type objects which are independent of their instantiations and depend solely on the structure of the program. The run-time entities are semantic transformers, defined for base types  $\underline{A}$ , which denote the many possibly different *abstract* interpretations of the same program.

$$\begin{aligned} ct &::= ct_1 \circ ct_2 \mid ct_1 \parallel ct_2 \mid ct_1 + ct_2 \mid rt \\ rt &::= \underline{A} \mid [\underline{A}, \underline{A}] \mid rt_1 \rightrightarrows rt_2 \end{aligned}$$

Generically, the program semantics of a programming language is expressed by a nondeterministic transition system  $TransSys\ a$ , where the variable  $a$  denotes a nonempty set of states. Each transition  $Rel$  is a binary relation between a state and its possible successors. From the transition system, we infer a control flow graph  $G = (V, E)$ , where the edge set  $E$  denotes transition relations and the node set  $V$  denotes the labels of the transition relations (associated to program states). Derivation of meta-programs is performed by traversing the control flow graph and applying the relational algebra to every connected subgraphs.

```
data Rel a = (Abstractable a) => Rel (a, String, a)
type TransSys a = [Rel a]
```

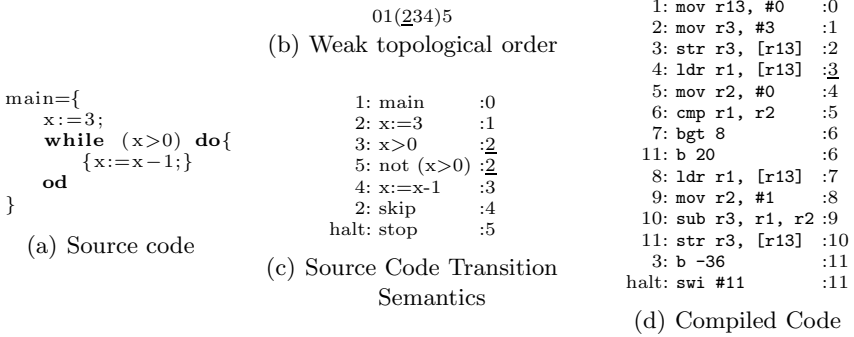
Our functional approach to data-flow analysis involves the computation of a system of fixpoint equations in the domain  $rt_1 \rightarrow rt_2$ . To this end, we apply the constructive design of a hierarchy of semantics of a transition system by abstract



interpretation [4]. Denotational semantics is defined as an abstract interpretation of the transition semantics. Hence, we abstract away from the history of computations by creating input-output functions which specify each particular effect. Thus, for each transition relation  $TransSys\ a$ , there is a relational abstraction  $RelAbs\ a$  which, in its turn, represents a subgraph of the meta-program. Since each subgraph has type  $a \rightarrow IO\ a$ , all meta-programs have the unified type  $a \rightarrow IO\ a$ . The use of the IO monad is required to support cache analysis of multi-core architectures, for which shared data memory is implemented using synchronizing variables  $MVar$  (see Section 5.2).

**type**  $RelAbs\ a = a \rightarrow IO\ a$

Consider the source code example in Figure 2(a). The corresponding weak topological order is given in Figure 2(b). Recursion is expressed by the label “2”, which is the *head* (underlined) node of the cycle **while**. Program semantics are non-deterministic because several statements can be referenced by the same label. In the example, two different actions can take place starting from label “2”: the execution enters the **while** loop body and the execution branches to label “3”; or the **while** condition evaluates to **False** and the execution is blocked with “halt”. Using the weak topological order, the labeled transition system is derived from the program syntax as described in Figure 2(c). Finally, Figure 2(d) shows the transition semantics of the compiled code.



**Fig. 2.** Example with recursion

Let us focus and examine the interpretation of the compiled code transition semantics. Each time two edges are connected by consecutive labels, we apply the sequential composition of the two corresponding subgraphs. The sequential composition ( $*$ ) of two relations  $T$  and  $R$  is defined by  $a(T * R)c$  iff there exists  $b$  such that  $aTb$  and  $bRc$ . In point-free notation, its type is  $T * R :: a \rightarrow c$ .

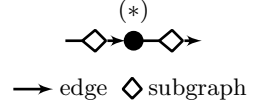
When two edges have the same source label and two different sink labels, we apply the parallel composition of the subsequent subgraphs. The parallel composition ( $/$ ) of two relations  $T$  and  $R$  is defined by  $(a, c)(T/R)(b, d)$  iff  $aTb \wedge cRd$ . Its type is  $(a, c) \rightarrow (b, d)$ .

$(*) :: (a \rightarrow IO\ b) \rightarrow (b \rightarrow IO\ c) \rightarrow (a \rightarrow IO\ c)$   
 $(f * g)\ s = f\ s \ggg g \ggg return$

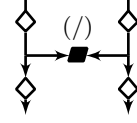
Example: `(mov r13, #0) * (mov r3, #3) * ...`

$(/) :: (a \rightarrow IO\ b) \rightarrow (c \rightarrow IO\ d) \rightarrow (a, c) \rightarrow IO\ (b, d)$   
 $(f / g)\ (s, t) = f\ s \ggg \lambda s' \rightarrow g\ t \ggg \lambda s'' \rightarrow return\ (s', s'')$

Example: `(b 20) / (ldr r1, [r13, #8]) * (mov r2, #1) *  
 (sub r3, r1, r2) * (str r3, [r13, #8])`



**Fig. 3.** Sequential composition



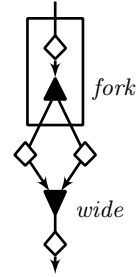
**Fig. 4.** Parallel composition

Interface adaptation is required prior and after parallel composition. Firstly, the *fork* function provides a copy of the previous subgraph output  $a$ , to the input of the two alternative subsequent paths. Secondly, when the output  $(a, a)$  of two parallel subgraphs are combined together, we apply the function *wide*.

$fork :: (Forkable\ a) \Rightarrow (a \rightarrow IO\ a) \rightarrow (a \rightarrow IO\ (a, a))$   
 $fork\ f\ s = f\ s \ggg \lambda s' \rightarrow$   
 $\quad branch\ s' \ggg \lambda b \rightarrow$   
 $\quad \text{if } b \text{ then } true\ s' \text{ else } false\ s\ s'$

$wide :: (Lattice\ a) \Rightarrow (a, a) \rightarrow IO\ a$   
 $wide\ (a, b) = join\ a\ b$

Example: `(fork bgt 8) * ((b 20) / (ldr r1, [r13, #8]) *  
 (mov r2, #1) * (sub r3, r1, r2) *  
 (str r3, [r13, #8])) * wide`



**Fig. 5.** Branching

Inside *fork*, it is necessary to instrument the contents of the *program counter* register so that the alternative paths can be taken. If the *branch* condition evaluates to True, we instrument the alternative path so that the evaluation to False is also taken into consideration. The converse applies if branch condition evaluates to False. Instances of type class *Forkable* provide the desired behavior.

The widening of subgraphs requires that program states constitute a lattice  $\langle L, \sqcup, \sqcap \rangle$ , where  $\sqcup$  is the *join* of program states and  $\sqcap$  is the *meet* of program states. Thus, for every program *State*  $a$ , the type variable  $a$  must implement an instance of the type class *Lattice*. Although the framework of abstract interpretation requires that the abstract domain is a complete lattice, the computation of *forward abstract interpretations* [5] only require the implementation of the *join* operator, by the fact that the static analyzer computes the least fixed point. Even though it is necessary to perform *backward abstract interpretations* for conditional instructions, such as “branch if greater” (*bgt*), this affects the values of the CPU registers only. In this special case, an implementation of the *meet* operator is required.

```

class Forkable a where
  branch :: a → a → IO Bool
  false  :: a → a → IO (a, a)
  true   :: a → a → IO (a, a)

```

```

class (Eq a, Ord a) ⇒ Lattice a where
  bottom :: a
  join    :: a → a → IO a
  meet    :: a → a → IO a

```

The analysis of loops is performed by the recursive point-free operator  $(+)$ . The interpretation of a loop corresponds to the reflexive transitive closure of the subgraphs that constitute the loop. In this way, we distinguish in the structure of the loop two subgraphs: the body ( $T$ ) and the branch condition ( $R$ ). The recursive composition  $(+)$  of two relations  $T$  and  $R$  is defined by  $a'(T+R)(a \mid a'')$  if  $aTa' \wedge a'Ra \vee aTa' \wedge a'Ra''$  and its type is  $a \rightarrow a$ . Further, we distinguish between the first loop iteration (dashed line in Figure 6) from others [8].

```

(+)      :: (Iterable a)
          ⇒ (a → IO a) → (a → IO a) → (a → IO a)
(f + t) s = t s ≫ λs' → if ¬ (fixed s')
                      then (f * (f + t)) (loop s')
                      else return (unroll s s')

```

Example:  $(\dots) * \text{wide} * ((\text{ldr } r1, [r13, \#8]) * (\dots) * (\text{fork } \text{bgt } 8) * (\dots) * \text{wide} + \text{b } -36)$

```

class Iterable a where
  fixed  :: a → IO Bool
  loop   :: a → IO a
  unroll :: a → a → IO a

```

Instrumentation of the analysis of loops is possible through the instantiation of the type class *Iterable*. The function *fixed* determines if the static analysis has stabilized at the entry point (*head*) of the loop. The function *loop* is

used to instrument the state of the analysis, for example, in order to find the minimal and maximal number of iterations of the loop. Finally, the function *unroll* finalizes the output state of the loop, which requires the subtraction of the effect of having the first loop iteration moved outside the loop.

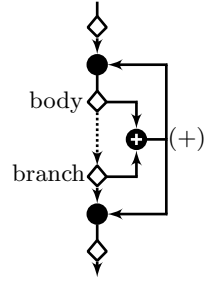


Fig. 6. Recursion

## 4 Fixpoint Semantics

For the purpose of data-flow analysis, program semantics are computed in fixpoint form, taking into account all possible program paths between two given program points. Each elementary path corresponds to a subgraph and is expressed by a semantic transformer of type *rt*. Using the meta-language, these semantic transformers are combined in order to derive a meta-program that corresponds to the *merge over all paths* (MOP) specification. Since the MOP solution is undecidable, we obtain a *minimum fixed-point* (MFP) by computing *joins* ( $\sqcup$ ) at merge points.

Two main advantages of using denotational definitions arise. The first is a consequence of the polymorphic definition of the meta-language that allows the computation of multiple effects for the same meta-program [12,9], provided the desired semantic transformers (the syntactical objects of a particular interpretation are hidden from the meta-language). The second advantage of the denotational meta-language is its natural support for interprocedural analysis. Indeed, instead of computing fixpoints by a symbolic execution of the program using an operational semantics of programs, we simulate program execution by computing the semantic transformers associated to blocks of instructions or procedures.

For static analysis purposes, program states *State a* are assigned to each graph node. Program states associate a *Label* according to the weak topological order, and an abstract evaluation context *Invs a*. Evaluation contexts are maps between program points and program values of type *Node a*, where *a* is the type variable denoting the domain of interpretation. To every graph node are adjoined two flags: the first indicates the state of the fixpoint iteration on that node; and the second tells if the node is inside a loop or not.

```
data State a = State (Label, Invs a)
type Invs a = Map.Map Int (Node b)
data Node a = Node (a, Bool, InLoop) deriving (Eq, Ord)
data InLoop = Yes | No deriving (Eq, Ord)
data Label = HeadL Int | NodeL Int | HaltL Int | Empty deriving (Ord)
```

The abstraction of the transition semantics is implemented by the type class *Abstractable*. For each transition *Rel a* is associated the right-image isomorphism *RelAbs a*. Thus, the denotational semantics *DenS a* maps the set of transition relations to their abstractions. The type class *Abstractable a* is used to apply a semantic transformer during fixpoint computation. That is, given a program state *State a* and an evaluation context *Invs a*, the semantic transformer *apply* reads the target label from the transition relation *Rel a*, then computes the effect given by *RelAbs (Invs a)* and, finally, stores it inside the target label.

```
class Abstractable a where
  apply :: (State a) → (Rel a) → (RelAbs (Invs a)) → IO (State a)
  label :: a → Label
```

The meaning of programs is taken from the least fixed point of the denotational semantics *DenS a*, where *a* is the domain of interpretation. For every pair of states *State a*, stored inside a transition relation *Rel a*, is associated a relational abstraction *RelAbs (Invs a)*.

```
type DenS a = Map (Rel a) (RelAbs a)
```

According to the Kleene first recursion theorem, every continuous functional  $F : L \rightarrow L$ , defined over the lattice  $\langle L, \subseteq, \perp, \top, \cup, \cap \rangle$ , has a least fixed point given by  $\bigcup_{\delta \geq 0} F^\delta$ , being  $F^\delta$  an ultimately stationary increasing chain ( $\delta < \lambda$ ). Given the *Lattice a*, continuous functionals *F* are denoted by *RelAbs a*. Fixpoint stabilization is reached when consecutive functional applications of *RelAbs a* produce the same output, i.e. the least upper bound of *Lattice a*.

The system of equations contained in  $DenS\ a$  is solved using the chaotic iteration strategy [2]. The chaotic iteration strategy consists in recursively traversing the control flow graph, labeled according to a weak topological order. Therefore, fixpoint iterations mimic the execution order which, in its turn, is expressed by the meta-language. In this way, the execution of a meta-program is in direct correspondence with its fixpoint.

During chaotic iterations, the data flow dependency in the program is taken into consideration in such a way that interpretation of sequential statements is made only once and in the right order. In the case of loops, it is sufficient to have stabilization at the entry point of the loop, for the whole loop to be stable. Note that the instantiation of the domain of interpretation  $a$ , allows the definition of different abstract interpretations for the same program.

The algorithm *fixpoint* used to derive meta-programs from denotational semantics is implemented by

```
type Fixpoint  $a = (Label, RelAbs\ a)$ 
fixpoint :: (Lattice  $a$ , Iterable  $a$ , Forkable  $a$ )
   $\Rightarrow DenS\ a \rightarrow Label \rightarrow Label \rightarrow Bool \rightarrow Fixpoint\ a \rightarrow IO\ (Fixpoint\ a)$ 
```

Given a denotational semantics  $DenS\ a$ , the *start* and *end* labels (*Label*), the flag used to control loop unrolling (*Bool*) and an initial fixpoint program, the algorithm returns the MOP representation of the denotational semantics. Starting with the relational abstraction  $f = id$ , the algorithm proceeds recursively throughout the following steps:

1. If the *start* and *end* labels are equal then return  $f$ ;
2. Determine the control flow pattern that follows the *start* label;
3. If the parsing of the denotation semantics has reached *halt* then return  $f$ ;
4. Else, if a sequential subgraph follows the *start* label, then obtain the relational abstraction  $f'$  associated with the subsequent label and recursively invoke *fixpoint* with  $f * f'$  as the actual meta-program;
5. Else, if multiple paths are available, the algorithm distinguishes between the recursive pattern and the fork pattern;
6. If the recursive pattern is found, compute the sub- meta-program *body* that corresponds to the body of the loop and the sub- meta-program *branch* that corresponds to the branch condition of the loop. Then, recursively evaluate multiple paths with  $f * (body + branch)$  as the actual meta-program;
7. If the fork pattern is found, obtain the branch condition *branch*, compute the meta-programs *true* and *false* which correspond to every pair of alternatives and, finally, recursively evaluate multiple paths with  $((fork\ test) * (true/false)) * wide$  as the actual meta-program;
8. When the evaluation of multiple paths terminates producing the meta-program  $f$ , recursively invoke *fixpoint* with  $f * f'$  as the actual meta-program;
9. When the algorithm returns, it is also available the last reached label.

The output of this process is a meta-program that computes by successive approximations an evaluation context ( $Invs\ a$ ) at every program points. The access

(*get*) and the update (*set*) of the evaluation context is accomplished by an instance of the type class *Container*.

A chaotic iteration comprises the following steps. The last computed state  $l$  for the program label  $at$  is passed as argument. Then, the previous computed state  $l'$  is retrieved from the evaluation context and joined with the actual state to produce a new state  $l''$ . If the fixpoint condition  $l'' \equiv l'$  verifies, then the fixpoint stabilizes for the program label  $at$ . For this purpose, the function *stabilizes* sets the fixpoint flag defined in the data type *Node* to *True*.

```

chaotic :: (Eq a, Lattice a) => Label -> Invs a -> a -> IO (Invs a)
chaotic at cert l = get cert at >>= \l' ->
  join l' l >>= \l'' ->
    if l'' == l'
      then stabilize cert at
      else set cert at l'' True

```

```

stabilize :: Invs a -> Label -> IO (Invs a)
stabilize cert at = let f (Node (s, c, _, l)) = Node (s, c, True, l)
  in return (adjust f (fromEnum at) cert)

```

```

class (Lattice b) => Container a b where
  get :: forall l -> (Enum l) => a -> l -> IO b
  set :: forall l -> (Enum l) => a -> l -> b -> Bool -> IO a

```

In order to perform timing analysis at machine level, our polymorphic fixpoint framework uses the same meta-program to statically analyze the microprocessor's registers file, cache and the pipeline. To this end, the type variable  $a$  is instantiated into the appropriate abstract domains. As an example, consider the microprocessor domain *CPU*. The fixpoint of an assembly program approximates microprocessor states *State CPU*, defined by the evaluation context *Invs CPU* at every program point *Label*.

Let *simulate* be the semantic transformer defined for the domain *CPU*. The static analysis method consists in the *lift* of this semantic transformer to the domain of the evaluation context *Invs CPU*. Finally, the function *transf* creates an instance of the function *apply* (defined in the type class *Abstractable*), which is the relational abstraction associated to the transition relation *Rel (State CPU)*. The functions *source* and *sink* simply return the states at the edges of the transition relation.

```

data CPU = CPU { memory :: Memory, registers :: Registers, pipeline :: Pipeline }
simulate :: Instruction -> CPU -> IO CPU

```

```

transf :: (Rel (State CPU)) -> Instruction -> RelAbs (State CPU)
transf i rel = \s -> let step = \cpu -> simulate i cpu
  eval = lift (sink r, source r) step
  in apply s rel eval

```

```

lift :: (Container (Invs b) b) => (Label, Label) -> (Rel b) -> (RelAbs (Invs b))
lift (after, at) f cert = get cert at >>= f >>= chaotic after cert

```

## 5 The Static Analyzer

The computation of the worst-case of program execution times is done compositionally, instruction by instruction. At this level of abstraction, the microarchitectural analysis employs abstract models of the microprocessor with the objective to compute a cycle-level abstract semantics in order to derive sound upper bounds for the execution times of an arbitrary basic block. Nonetheless, while abstract interpretation techniques are capable to determine the WCET for all possible program paths, the calculation of the worst-case program path requires an additional *path analysis* [11].

In resume, WCET estimation is a process that requires three main steps: (1) the *program flow analysis* is used to obtain flow information from the program source code without manual intervention; (2) the *microarchitectural analysis* computes safe bounds on the execution times for every instruction in the compiled program and is further divided into *value analysis*, *cache analysis* and *pipeline analysis*; (3) the *ILP calculation* combines program flow and atomic execution times into a WCET estimation using algebraic constraints. Due to space limitations, here we will describe only the implementation guidelines of the two first steps.

Figure 7 shows how the static analyzer combines the several analyses in order to provide a WCET-aware environment for software development. The additional Back-Annotation component is responsible for the annotation of the source code with the local execution times computed during pipeline analysis. For this purpose, compiler debug information is used (see Figure 1(b)).

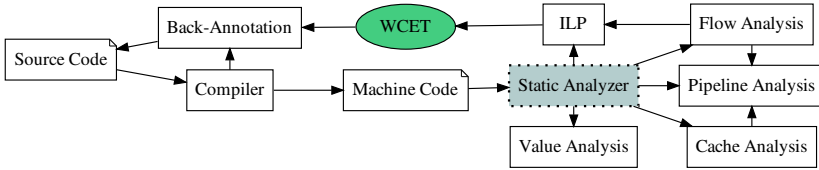


Fig. 7. WCET analysis tool chain

### 5.1 Program Flow Analysis

Path information is essential for tight WCET calculation, namely to identify *false paths*, i.e. paths that can never be taken, and to count the number of *iterations* in loops. Since the calculation of the number of iterations in loops is in general equivalent to the halting problem, we need to consider *all* possible executions and reduce the computational cost by using abstractions in order to compute the maximal number of iterations in loops [7]. Safe path information is obtained automatically by abstract interpretation as an instrumented version of the value analysis by means of the type class *Iterable*.

Instrumented interpretation requires the definition of a new state type called *IState*. It includes two new evaluation contexts: *Loops* and *Paths*. The first associates the number of iterations to program points and the second associates path feasibility to program points.

```
data State b = State (Label, Invs b)
                | IState (Label, Invs b, Paths, Loops) deriving (Ord, Eq)
type Loops = Map Int Loop
type Paths = Map Int Path
data Loop = Count Int | BottomL deriving Eq
data Path = TopP | BottomP deriving Eq
```

Each time a recursive branch is taken, we apply the function *loop* defined in the type class *Iterable* that increments loop iterations by one. After reaching the fixpoint of the same loop, loop iterations are unrolled by a factor of one. This is a consequence of the algorithm used to derive the meta-program, which explicitly unrolls the first iteration to distinguish the first iteration from all others.

```
loop (IState (after, cert, paths, loops, i))
  = let label = fromEnum after
      loops' = mapWithKey (\k l → if k ≡ label then succ l else l) loops
      cert'  = mapWithKey (\k l → if k ≡ label then inLoop l else l) cert
  in return (IState (label, cert', paths, loops', i))
```

For the detection of false paths we instrument the value analysis using the function *false* defined in the type class *Forkable*. Every time a *false* path is taken at given program point, we compute the meet ( $\sqcap$ ) between the previous computed *Path* at that point with *BottomP*. In this way, after reaching the fixpoint, all program points instrumented with *BottomP* will be considered non-executable paths.

## 5.2 Microarchitectural Analysis

Our target platform is based on ARM microprocessor instruction set [13], the most popular platform for embedded real-time systems. It is a 32-bit reduced instruction set computer (RISC) instruction set architecture (ISA) with 16 registers, 4 state bits and 32-bit addressable memory. The considered cache replacement policy is the *Least Recently Used* (LRU) and pipelining is done in five stages.

**Value Analysis.** The objective of value analysis is to obtain sound approximations of the set of values, inside the registers bank and data memory, that may be produced at some program point. For this purpose, the interval domain (*Interval*) is chosen as abstract domain of interpretation.

The registers bank *Registers* consists on 16 general purpose registers (*R0*, *R1*, ..., *R15*) and one current program status register (*CPSR*). The domain of data memory values *DataMem* 32-bit address values (*Word32*) to memory values *MemValue*.



```

type Interval      = (Word32, Word32)
type Registers     = Array RegisterName Register
data Register      = BottomR | Pc Word32 | Status Word32 |
                      Interval Interval | TopR deriving (Eq)
data RegisterName = R0 | R1 | ... | CPSR deriving (Enum, Eq,Ix, Ord, Read)

```

There are some exceptions concerning the abstract domain of registers. The abstract domain *Status* of program status register *CPSR* contains a concrete value of type *Word32*. Similarly, for the remaining registers stack pointer *R13*, link register *R14* and program counter *R15*, the abstract domain *Pc* contains concrete values. The design of the register domain in this way allows the chaotic fixpoint algorithm to *simulate* the abstract interpretation of the program by taking into account the data flow dependency in the program, using namely the contents of *R15*. On the other hand, the static analyzer is not able to compute safe bounds on the *concrete* registers.

**Cache Analysis.** Cache analysis is based on the notion of *age* of a memory block [16]. Two simultaneous static analysis are performed: one called *May analysis* that approximates ages from below and classify definite cache misses, i.e. the set of memory blocks that *may not* be in the cache at a program point; another called *Must analysis* that approximates ages from above and classify definite cache hits, i.e. the set of memory blocks that *must* be in the cache at a program point.

```

data Memory
  = Memory { datam :: DataMem, must :: MVar Main, may :: MVar Main
            , mustL1 :: L1Box, mayL1 :: L1Box, mustL2 :: L2Box
            , mayL2 :: L2Box } deriving (Eq)

```

Our memory model supports multi-core design where each processor core has a private first-level cache *L1Box* and a second-level cache *L2Box* shared across the processor cores. The L1 caches implement the *MESI* cache coherence protocol using the synchronous channel of type *Request*. Each elementary *Cache* is a fully-associative instruction cache where address blocks *CachedOp* are stored in along *n* cache lines (*Int*). The domain of concrete cached values *OpCode* holds either the undefined element *BottomW* or a *Word32* instruction *opcode*.

```

data L2Box      = L2Box (MVar Cache)
data L1Box      = L1Box { cache :: Cache, tc :: Chan Request, oc :: Chan Request }
type Cache      = Map Int [CachedOp]
type CachedOp   = (Address, (OpCode, MESI))
data OpCode     = OpCode Word32 | BottomW deriving (Eq)
data Request    = SnoopOnWrite CachedOp | SnoopOnRead CachedOp

```

The abstract domain [*CachedOp*] is the powerset of concrete values. Cache analysis is based on the notion of *age* of a memory block, i.e its position *p* inside the cache set ( $p \leq n$ ). The join of abstract cache states, specified in the type class *LRUCache*, is parameterized by two functions: one compares the ages of

two memory blocks ( $>$  or  $<$ ) and the second specifies the list operation (*union* or *intersect*), respectively for the May and Must analysis.

```
class LRUCache a where
  bottomLRU    :: IO a
  mergeBoxLRU :: a → a → (Int → Int → Bool) →
    ([Address] → [Address] → [Address]) → IO a
```

**Pipeline.** Pipeline analysis is based on the notion of number of *processor cycles* needed to completely process an instruction inside pipeline [15]. The timing model of pipeline analysis allows overlapped execution of instructions by dividing the execution of instructions into a sequence of five pipeline stages denoted by *Stage*, and by simultaneous processing three instructions. Each stage represents one phase of instruction execution: *fetch* (*FI*), *decode* (*DI*), *execute* (*EX*), *memory access* (*MEM*), and *write back* (*WB*) [15].

```
data Stage = FI | DI | EX | MEM | WB
```

The *Pipeline* consists in a sequence of pipeline states *PState* which depend on several components: the next program counter to fetch from the instruction cache (*Word32*), the internal program status register (*Word32*), the *Registers* file and the *Memory*, and finally the coordinates vector *Coord*. The abstract state of each task inside the pipeline *AbsTaskState* is a tuple consisting in the elapsed execution count (*Int*), the current *Stage* of the task and the actual computation stored in *TaskState*.

```
data Pipeline = Pipeline [StateVector]
data PState   = PState (Word32, Word32, Registers, Memory, Coord)
data Coord    = Coord (Vec3 (AbsTaskState))
data AbsTaskState = AbsTaskState (Int, Stage, TaskState)
data TaskState = Ready Task | Fetched Task Stubs | Decoded Task Stubs |
  Stalled Reason Task Stubs | Executed Task Stubs |
  Done Task deriving (Eq)
type Stubs     = [(RegisterName, Register)]
type Task      = (Instruction, Word32, Word32, Registers, Memory)
type AbsTask   = Int → TaskState → IO AbsTaskState
type FunArray  = Vec3 AbsTaskState
```

Successive states of each element in coordinates vector are obtained by a transfer function *AbsTask* that receives as arguments the actual execution count and a *TaskState* to produce a new *AbsTaskState*. This functional approach to pipelining is based on the instantiation of functions *AbsTask* that, using pattern matching on the current stage (*Ready*, *Fetched*, *Decoded*, *Stalled*, *Executed* and *Done*), update the execution count and the internal data (*Stubs*) stored in the pipeline.

At the higher level of program semantics, for each *Instruction* we apply the function *simulate* in order to update the state of the *CPU*. Recursive calls to the function *step* are made until the stage of the tasks changes to *Done*. Each step involves the instantiation of a *FunArray*, to which is passed the current *PState*. In each step is also necessary to join the program counter and the program

control status stored in the pipeline *Stub* with correspondent top-level data in the pipeline *PState*. When branch instructions are completed, the pipeline is flushed so that the next fetched instruction corresponds to the new program counter.

```

simulate :: Instruction → CPU → IO CPU
step      :: Instruction → StateVector → IO StateVector
next      :: Instruction → StateVector → FunArray
apply     :: StateVector → FunArray → IO StateVector

```

## 6 Conclusions

This paper introduced calculational design of a WCET static analyzer implemented in Haskell. The abstract interpretation is defined in terms of a two-level denotational meta-language with the purpose to generalize fixpoint computations using the reflexive transitive closure of different transition function systems. Very high-level descriptions of program semantics are obtained in the point-free notation and easily implemented in the highly declarative programming language Haskell. In this way, fixpoints correspond to an algebraic point-free version of the program, which is evaluated by a functional static analyzer.

The main advantage of using a functional approach is closely related with the calculational method proposed in [5], used to induce abstract interpretations from the concrete semantics as formal specifications. With such method, we start with the Haskell's standard denotational interpretations of the assembly programming language, and then apply the higher-order Galois connections framework in order to induce, although not in a mechanized way, abstract interpretations which have a straightforward implementation in Haskell.

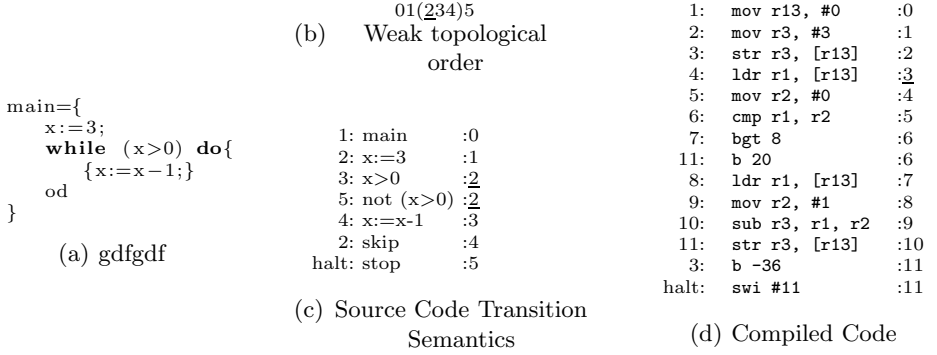
## References

1. AbsInt. Angewandte informatik., <http://www.absint.com/pag/>
2. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Pottosin, I.V., Bjorner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
3. Brassel, B., Christiansen, J.: Denotation by Transformation. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 90–105. Springer, Heidelberg (2008)
4. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science* 6 (1997)
5. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam (1999)
6. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: *Proceedings of the Second International Symposium on Programming*, Dunod, Paris, France, pp. 106–130 (1976)
7. Ermedahl, A., Gustafsson, J.: Deriving annotations for tight calculation of execution time. In: Lengauer, C., Griebel, M., Gorlatch, S. (eds.) *Euro-Par 1997*. LNCS, vol. 1300, pp. 1298–1307. Springer, Heidelberg (1997)

8. Ferdinand, C., Martin, F., Wilhelm, R., Alt, M.: Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.* 35, 163–189 (1999)
9. Jones, N.D., Nielson, F.: Abstract interpretation: a semantics-based tool for program analysis. In: *Handbook of Logic in Computer Science: Semantic Modelling*, vol. 4, pp. 527–636. Oxford University Press, Oxford (1995)
10. Lacan, P., Monfort, J.N., Ribal, L.V.Q., Deutsch, A., Gonthier, G.: ARIANE 5 - The Software Reliability Verification Process. In: *DASIA 1998 - Data Systems in Aerospace*. ESA Special Publication, vol. 422 (May 1998)
11. Li, Y.-T.S., Malik, S., Wolfe, A.: Cache modeling for real-time software: beyond direct mapped instruction caches. In: *IEEE Real-Time Systems Symposium*, pp. 254–263 (1996)
12. Nielson, H.R., Nielson, F.: Pragmatic aspects of two-level denotational meta-languages. In: Robinet, B., Wilhelm, R. (eds.) *ESOP 1986*. LNCS, vol. 213, Springer, Heidelberg (1986)
13. Patankar, V.A., Jain, A., Bryant, R.E.: Formal verification of an arm processor. In: *Twelfth International Conference On VLSI Design*, pp. 282–287 (1999)
14. Plazar, S., Lokuciejewski, P., Marwedel, P.: A Retargetable Framework for Multi-objective WCET-aware High-level Compiler Optimizations. In: *Proceedings of The 29th IEEE Real-Time Systems Symposium (RTSS) WiP*, Barcelona, Spain, pp. 49–52 (December 2008)
15. Schneider, J., Ferdinand, C.: Pipeline behavior prediction for superscalar processors by abstract interpretation. *SIGPLAN Not.* 34, 35–44 (1999)
16. Wilhelm, R., Wachter, B.: Abstract Interpretation with Applications to Timing Validation. In: Gupta, A., Malik, S. (eds.) *CAV 2008*. LNCS, vol. 5123, pp. 22–36. Springer, Heidelberg (2008)

## A Example

Here we show a simple example of two cores running the same assembly code. The source code example written in a programming language is a C-like language is shown in Figure 8(a). The weak topological order is given in Figure 8(b) and the labeled transition system of source code statements is shown in Figure 8(c). The labeled transition system of the compiled assembly code generated by our framework is shown in Figure 8(d).



**Fig. 8.** Simple example with recursion

The meta-program created by the fixpoint generator is:

```
(mov r13, #0)*(mov r3, #3)*(str r3, [r13])*(ldr r1, [r13])*(mov r2, #0)*(cmp r1, r2)*
fork bgt 8)*((b 20)/(ldr r1, [r13, #8])*(mov r2, #1)*(sub r3, r1, r2)*(str r3, [r13, #8]))*
wide*((ldr r1, [r13])*(mov r2, #0)*(cmp r1, r2)*(fork bgt 8)*((b 20)/(ldr r1, [r13, #8])*
(mov r2, #1)*(sub r3, r1, r2)*(str r3, [r13, #8]))*wide+(b -36))*(swi #11)
```

The value analysis of the register file at the last program point (*halt*) is:

-----Certificate-----											
Program point: halt Iterations 1											
-----Registers-----											
-----											
r0=	_	r4=	_	r8=	_	r12=	_				
r1=	(0,3)	r5=	_	r9=	_	r13=	(0,0)				
r2=	(0,1)	r6=	_	r10=	_	r14=	_				
r3=	(0,3)	r7=	_	r11=	_	r15=	72				
cpsr= 536870912 (N=0 Z=0 C=1 V=0)											
-----											

The value analysis of the data memory at the last program point is (the address of the variable *x* is 0x00):

```

+-----Certificate-----+
| Program point: halt Iterations 1 |
+-----+-----+
|                               Cache                               |
+-----+-----+
+===== Data Cache =====+
0x00: (0,3)
      : _|_
      : _|_
      : _|_
      : _|_
      : _|_
...

```

The cache analysis of the two level (L1 of size 5 and L2 of size 15) instruction cache is (addresses on the left):

```

+-----Certificate-----+
| Program point: 14 Iterations 1 |
+-----+-----+
|                               Memory                               |
+-----+-----+
0X00 : _|_
0X04 : _|_
0X08 : _|_
0x12 : _|_
0X16 : _|_
0X20 : _|_
0X24 : _|_
0X28 : 3830255616 read hit on L2
0X32 : _|_
0X36 : _|_
0X40 : _|_
0X44 : _|_
0X48 : _|_
0X52 : 3818921985 read hit on L1
0X56 : 3762368514 read hit on L1
0X60 : 3831312384 read hit on L1
0X64 : 3925934044 read hit on L1
0X68 : 4009754635 read hit on L1
...

```

The last pipeline state at the last program point is:

PState: next fetch=72; blocked=					
n	k	Next stage	State	Pc	Blocked
1	0	DI	Fetch: ldr r1, [r13]	32	
0	1	FI	Ready: nop	0	
6	2	WB	Done: swi #11	72	

When the assembly program runs on a single core, the calculated WCET is 267 cpu cycles. When the assembly program runs on a dual core, the WCET may be different on one of the cores due to the concurrent execution of one thread per core. This This is explained by the fact that the same set of instructions are part of the same assembly program, which may affect the contents of the shared cache on level L2. Additionally, the cache coherece protocol MESI will detect some *dirty* addresses in private L1 caches, which force new fetches from the upper memory levels and, consequently, more cache miss penalties will be taken into account.

# Building a Faceted Browser in CouchDB Using Views on Views and Erlang Metaprogramming


Claus Zinn

Dept. of Linguistics  
University of Tübingen  
`claus.zinn@uni-tuebingen.de`

**Abstract.** Consider sets of XML documents where documents from the same set adhere to the same schema, and documents from different sets adhere to a different schema. All documents describe language resources and tools, but as their schemas differ so differ their use of descriptors and the values they can hold. The collection of metadata documents and schemas is open and can get extended anytime. This paper describes a solution to the problem of storing all documents in a single database and making them accessible to naive users to easily identify language resources and tools according to their needs and interest. The proposed storage solution makes use of the document-based database CouchDB; for easy access, we propose a combination of faceted search and full-text search, allowing users without intricate knowledge about metadata descriptors to explore all documents in a systematic manner. Faceted search is entirely bootstrapped using CouchDB views and meta-views that we meta-programmed in Erlang given a declarative facet specification.

## 1 Introduction and Background

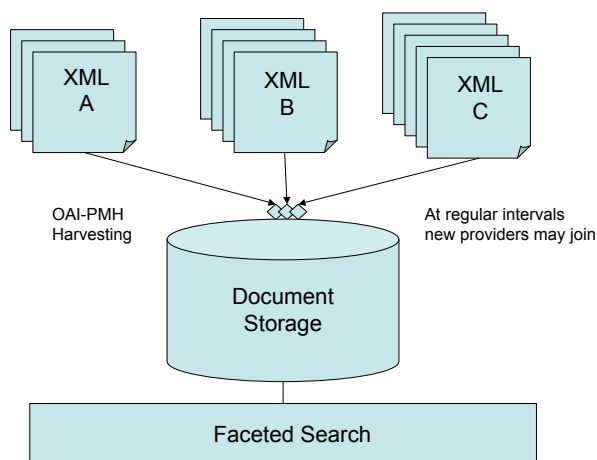
While there is a technical and commercial infrastructure in place to manage scientific publications, there is no systematic management of the underlying research data. In recent years, there has been an increasing pressure from funding agencies on institutions and individual researchers alike to describe their research data with metadata and to make such data public so that it can be easily accessed by the scientific community. Such an infrastructure would make it easier for researchers to reproduce results over identical data sets with an identical or different research method; it increases scientific quality and fights fraud in science; and it helps avoiding unmeant duplication of research work.

The NaLiDa  project, funded by the German Research Foundation, aims at contributing parts of this infrastructure for languages resources (corpora, lexica *etc.*) and software tools (part-of-speech taggers, parsers *etc.*), and supporting partners in the scientific community with infrastructure building, metadata

---

<sup>1</sup> NaLiDa is the acronym for the German “**N**achhaltigkeit **L**inguistischer **D**aten” (sustainability of linguistic data). See <http://www.sfs.uni-tuebingen.de/nalida/>.

management and storage. It assists institutions to systematically describe their research with metadata, and encourages them to expose their metadata holdings in terms of XML-based documents. For this, it encourages the use of the *Open Archives Initiative Protocol for Metadata Harvesting* (OAI-PMH) so that NaLiDa can index or catalogue them.



**Fig. 1.** General Setting

The problem to be solved is depicted in Fig. 1. Documents from different partner institutions will usually adhere to different schemas, and thus have different sets and uses of metadata field descriptors and corresponding value ranges. Moreover, the holdings of an institution will change as new metadata descriptions for language resources or tools are being created, or existing ones modified. Document sets will need to be harvested at regular intervals to ensure high synchronicity between the central storage and its individual document sources. In light of considerable data heterogeneity, an easy and effective method to access and browse all data is needed.

In this paper, we describe the design and implementation of a faceted search component to give naive users, users without intricate knowledge about metadata structures and descriptors, a uniform and multi-dimensional access to the aggregation of all metadata documents. Our approach is based upon the document-based database CouchDB; it makes use of metaprogramming techniques to bootstrap the faceted search's back-engine entirely by generating Erlang code from a declarative facet specification. CouchDB's map-reduce framework is used in two phases, with the second phase introducing a kind of *views of views*.



## 2 Faceted Browsing

### 2.1 Motivation

Faceted search seems well-suited for naive users to explore large data sets with a small but informative set of facets. The method is becoming increasingly popular in many commercial websites. It allows customers to identify products along many dimensions by selecting those facets that are more important to them than other criteria. The presentation of facets, together with their value range and the number of corresponding product items, gives users a good overview of the structure and content of the search space. In fact, many users *learn* the main criteria to systematically select a given item, and without such structured search, they would be less able to make an informed choice.

Metadata descriptions in the area of language resources and tools can be very detailed. Given the complexity of the field, there is a large variety in the usage of metadata field descriptors and their structural organisation. While most of this information is of little use for naive users, there are some pieces that matter for most users, and which can be used for faceted navigation. In part, facet selection is also governed by the search for a common denominator across document collections; in fact, this search will yield a rather small set of (semantically similar) metadata fields that can potentially serve as facets. In the given domain, we found the following information to be shared by a majority of documents: the organisation that has created the resource or tool (with values being the organisation names); the language that is being described or processed (language names or codes such as ISO 639-2), the type of resource (corpus, lexicon, tool *etc.*), and modality information (written, spoken *etc.*).

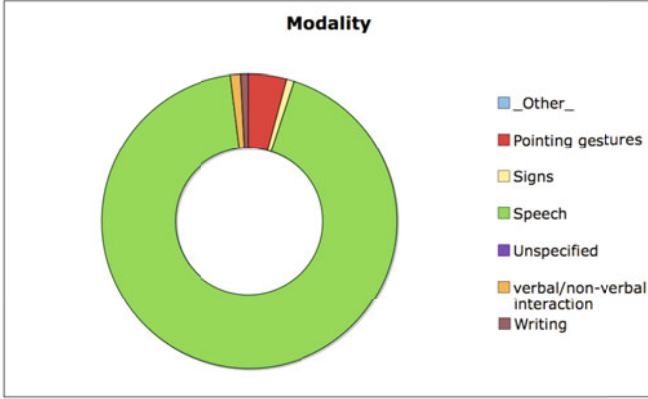
We also found information that is only commonly shared across well-defined document subsets. Documents describing language tools, for instance, make frequent reference to the type of tool (spell checking, named entity recognition *etc.*), its lifecycle status (development, production *etc.*) as well as input and output formats. Here, the introduction of *conditional* facets would help users to cluster the corresponding document subspace once it is entered. Conditional facets, thus, serve as additional navigational aid once the context permits their use; moreover, they also keep performance issues in check.

While the sequel of our discussion will stick to the application context “language resources”, it should be clear that our approach is applicable to document collections of other domains.

### 2.2 Theoretical Background

Let  $F_1, \dots, F_n$  be facets with their respective values ranges

$$\{f_{11}, \dots, f_{1n}\} \dots \{f_{n1}, \dots, f_{nm}\}.$$



**Fig. 2.** Indexing of all documents (*facetification*)

A metadata document must be described by at least one facet-value pair, otherwise it will not be accessible via faceted search. Moreover, a document can be described by more than one value  $f_{ij}$  for a given facet  $F_i$ <sup>2</sup>

Fig. 2 describes the result of indexing a document set in terms of the facet “modality” and the values it can take. Each ring segment represents a non-empty set of documents where a document can appear in more than one ring segment. The initial view in faceted search (prior to any facet selection) can be read directly from these data rings.

Once a facet-value pair  $f_{ik}$  is selected, the corresponding document set  $f_{ik}$  must be intersected with each of the other subsets of  $F_j$  with  $1 < j < n$ ,  $j \neq i$ . In other words, the document set of the ring segment  $f_{ik}$  must be intersected with the document sets of all segments of all rings other than  $F_i$ . When users select a facet  $F_i$  with value  $f_{ik}$  and a facet  $F_j$  with value  $f_{jl}$ , then we first build the intersection between the two corresponding document collections; given the resulting intersection is non-empty, it must be intersected with all ring segments of all rings other than  $F_i$  and  $F_j$ .

### 3 CouchDB and Its Map-Reduce Framework

#### 3.1 Design and Engineering Choices

Design and engineering choices were driven by the following requirements:

- cope with metadata heterogeneity, given that documents will adhere to different schemas each defining its own structured set of descriptors and values;

<sup>2</sup> A multimodal corpus, for instance, may be described with the facet “modality” and its values “gesture”, “sign language” and “spoken language”.

- preserve the original format of all metadata descriptions, and consider storing primary data in addition to the metadata describing it;
- handle regular additions to document storage with only incremental update for document access;
- provide effective and user-friendly access to all documents in the most efficient manner; and
- use a REST-based approach [9] to make data storage read & write web-accessible.

The document-based database CouchDB [1] fulfills all the requirements. Its schema-less database design permits the inclusion of arbitrarily structured documents into the database. Through CouchDB's document attachment mechanism, the original metadata format can be preserved, and primary data such as language recordings, transcriptions, experimental data *etc.* can also be associated with the metadata describing it. CouchDB's map-reduce framework promises incrementality and scalability. CouchDB also features a REST-based interface for document uploading, downloading and querying. Moreover, documents describing the faceted browser's GUI (in terms of html, css and JavaScript files) can be attached to so-called *CouchDB design documents* so that CouchDB's REST interface makes CouchDB playing the role of a GUI server, so to speak. In addition, a port to Lucene makes the provision of full-text search to all documents and to documents attached to them easy to implement.

CouchDB is written in Erlang. Its document-oriented database can be queried using the default language JavaScript or CouchDB's native language Erlang<sup>3</sup>. The indexing or querying of a database's documents is defined in terms of *views* whose definitions must also be stored in the database (again attached to the database's area for design documents). In contrast to traditional databases, thus, CouchDB expects all queries to be anticipated and defined in advance. Once a query is executed (that is, a view is computed), its result is stored. In case, the document base changes, all views are recomputed by only taking the incremental change into account.

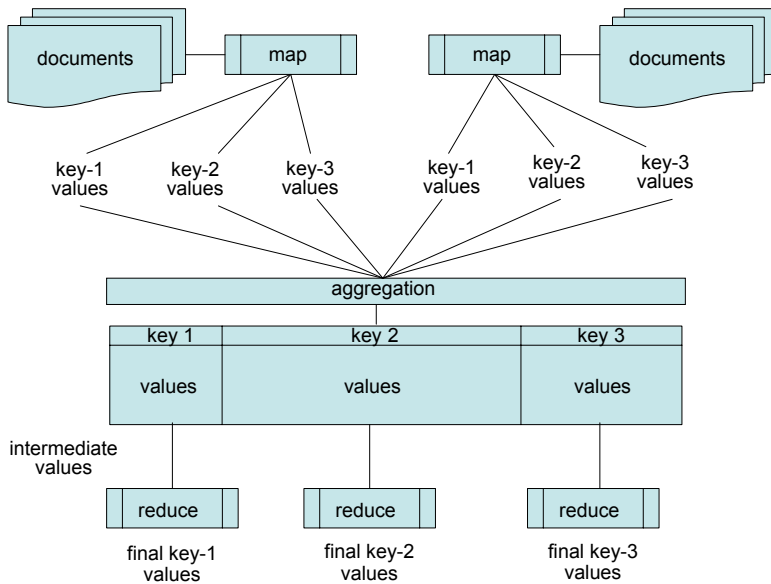
### 3.2 Map-Reduce Framework

Queries in CouchDB are *views*, which are computed following a *map-reduce* framework [4]. The framework is depicted in Fig. 3.

*Map.* A CouchDB view is defined in terms of a *map* function that is required to be *referentially transparent*. That is, given a document, the function will always emit the same key-value pairs. The document indexing process can thus be run in parallel as the indexing of one document does not depend on the indexing of other documents. Moreover, the map process can be realized as an incremental process. As more documents come in, the result table of emitted values can be extended to complement the index.

---

<sup>3</sup> The CouchDB community is developing support for other programming languages to be used for indexing and querying, such as PHP, Ruby, Python, or Haskell.



**Fig. 3.** General map-reduce framework

The map function to implement the elementary view representing the facet “organisation” will, for instance, emit the key/value pair (“University of Leipzig”, 1), given that the corresponding document mentions “University of Leipzig” in the respective metadata field given its schema definition.

*Reduce.* The map function of a CouchDB view can be complemented by a *reduce* function. It takes as input the table of emitted values with identical keys as generated by the map function, and aggregates them. A simple aggregation to sum up the values associated with the same key can be implemented in JavaScript as

```
function(keys, values) { return sum(values); }
```

In the given example, we would obtain an aggregated view of metadata documents with regard to the organizations they originate from, *e.g.*, [(“University of Leipzig”, 120), (“University of Tuebingen”, 180), ...].

A *reduce* function must be referentially transparent, but also commutative and associative. In particular, it must be possible for a reduce function to be called not only with table entries resulting from the map process, but also with intermediate values computed by a prior reduce process.

## 4 Implementation

The implementation of a faceted browser for metadata documents on language resources and tools is divided into the main phases for document ingestion, document indexing and data curation, the generation of document indices and views for faceted search, and the use of these views in the graphical user interface.

### 4.1 Document Ingestion

All documents harvested via OAI-PMH are being automatically validated against their schema and, given successful validation, ingested into CouchDB. The ingestion process converts XML documents into JSON structures (a lightweight data-interchange format, see <http://www.json.org/>), generates a unique document id, adds extra metadata to mark a document's origin (its data provider), makes its schema reference accessible at `doc.schema`, and adds an ingestion timestamp. The enriched JSON document is then added to the database of all documents, together with the original XML document as CouchDB document attachment. CouchDB also offers a versioning mechanism that preserves previous document versions when documents with existing ids are being uploaded.

### 4.2 Document Indexing with Map-Reduce

The document indexing process attacks the problem of data heterogeneity given that documents may adhere to different schemas. In CouchDB, indexing is naturally defined in terms of a *map* and, for statistical purposes, a *reduce* function. The map function indexes all documents along the dimensions defined in the facet specification for both unconditional and conditional facets. Fig. 4 gives an abstract description using JavaScript as view definition language. During view computation, the map function is called for each CouchDB document `doc` (which

```
function(doc) {
  switch( doc.schema )
  {
    case "<reference_to_schema_a>":
      if ( <tree_has_node>
        ) {
        emit(<path_to_node_val>, 1);
        break;
      }

    case "<reference_to_schema_b>":
      [...]
    [...]
  }
}
```

Fig. 4. JavaScript template for CouchDB view

```

{ "facet"      : "modality",
  "pathInfos" : [
    { "schema": "http://catalog.clarin.eu/...:cr1:p_1290431694580/...",
      "path"   : "doc.CMD.Components.TextCorpusProfile...",
    },
    { "schema": "http://catalog.clarin.eu/...:cr1:p_1290431694579/...",
      "path"   : "doc.CMD.Components.LexicalResourceProfile..."
    },
    ...
  ]
}
{ "facet"      : "language",
  "pathInfos"  : [ ... ]
}
[...]
```

**Fig. 5.** Declarative specification of facets (simplified)

is represented in JSON). Depending on the value of the document's schema (`doc.schema`), it checks whether a certain node in its document tree exists, and if this is the case, *emits* the node's value as facet value.

Initially, we have coded map functions manually. The resulting code was adapted whenever a document schema was newly defined or modified. Coding map functions for indexing is tedious and prone to error, so we moved toward the automatic generation of views from a declarative facet specification using JavaScript. Fig. 5 sketches the specification of facets. For each facet, we specify the path where to find the facet's value in the document, given the document's schema. We have written JavaScript code that uses the specification as input, and generates code that mirrors the code structure given in Fig. 4. Given that JavaScript does not support macros, we constructed code through string manipulations, and uploaded (via CouchDB's REST interface) the resulting code into a CouchDB design document.

We have used JavaScript for this because it is the default language for the definition of CouchDB views. JavaScript also natively supports JSON (CouchDB's native document format) and also offers a library with XPATH<sup>4</sup>-like functionality to navigate through JSON structures.

### 4.3 Data Curation and Conversion of Views to Documents

The map functions that we meta-programmed using the information in the facet specification are uploaded into the CouchDB database that also contains all metadata documents. Each map function gives thus a view of the document space in terms of the facet it represents. An analysis of the views showed a large variability for many facet values. The computation of the view "organisation", for instance, yields a table that relates organisation names with document sets

<sup>4</sup> See <http://www.w3.org/TR/xpath/>.

carrying this information; table rows are redundant, however, when different names (e.g., “MPI for Psycholinguistics” and “Max-Planck-Institute for Psycholinguistics” denote the same organisation. Here, data curation is necessary. We have used the results of our views analysis to devise *curation tables* that map given names to preferred names. In case of the facet “organisation” we have also made use of a database that is maintained by the German National Library (DNB) and distributed via the Linked Data initiative (see <http://linkeddata.org/>). The DNB database assigns unique identifiers to institutional bodies, together with their preferred name(s) and name variants. Our curation tables are manually updated whenever we detect an unknown facet term that appears to be semantically equivalent to an existing one.

Faceted search needs to be defined in terms of the document indexing established in the first map-reduce cycle. However, CouchDB’s map-reduce framework

```

fun ({Doc}) ->
  case proplists:get_value(<<"docType">>, Doc) of <<"docIndex">> ->

    % get hash tables for each of the facets
    {CountryHash} = proplists:get_value(<<"country">>, Doc, {}),
    {LanguageHash} = proplists:get_value(<<"language">>, Doc, {}),
    <other hashes>

    % for each key in the given hash table
    lists:foreach(fun (CountryItem) ->
      DocSet = proplists:get_value(CountryItem, CountryHash),
      DocSetSize = ordsets:size(DocSet),
      if DocSetSize > 0 ->
        % emit the number of documents available for each of the facet's values
        Emit(CountryItem,
          {[<<"facet">>, <<"_total_">>],
           [<<"value">>, <<"_total_">>],
           [<<"docs">>, DocSet]}},
          <other hashes>);
      % emit the intersections
      lists:foreach(fun (LanguageItem) ->
        Intersection = ordsets:intersection(proplists:get_value(LanguageItem,
                                                                    LanguageHash),
                                              proplists:get_value(CountryItem,
                                                                    CountryHash)),
        case Intersection == [] of false ->
          Emit(CountryItem,
            {[<<"facet">>, <<"language">>],
             [<<"value">>, LanguageItem],
             [<<"docs">>, ordsets:size(Intersection)]});
        _ -> ok
        end
      end,
      proplists:get_keys(LanguageHash)),
      <other intersections for other facets[...]>
    true -> ok
  end
end,
proplists:get_keys(CountryHash));
_ -> ok
end
end.

```

**Fig. 6.** Exemplary Erlang code for the facet “country”

is defined in terms of documents and it is thus not possible to define views on views, at least not directly. To re-use the result of document indexing for faceted search, we converted views into documents containing the indices. The conversion also included data curation so that documents associated with two semantically identical keys were merged into a single set using the preferred key<sup>5</sup>.

We implemented the conversion process in JavaScript. We queried each view and added the resulting JSON structure to another JSON structure containing all indices modulo data curation. This large JSON data structure represents a hash table of hash tables. The outer hash table gives access to the facets (*e.g.*, “organisation”), and the inner hash table to all the values a chosen hash can take. The inner hash key “University of Tuebingen”, for instance, is associated with all documents that share this piece of information. The new index (of type “docIndex”) is stored into a special purpose database to also hold all views to implement faceted search. We have created such index files for each document selection obtained by the respective data providers.

#### 4.4 Faceted Search with Map-Reduce

Fig. 6 displays an Erlang code fragment for the view “country”. First, we test whether the document to be processed is of type `docIndex`. If this is the case, we retrieve from the document the hashes for each of the facets. Then, for each key `CountryItem` of the hash `country`, we do the following: we emit the number of documents indexed with `CountryItem`; then we build the intersections between the set of all documents indexed with `CountryItem` with all sets described by all keys of all other hashes.

The map function for the given view “country” emits a table of key-value pairs comprising, for instance, the following entries:

Key	Value
"Germany"	{facet: "_total_", value: "_total_", docs: [d1, d2, ...]}
"Germany"	{facet: "language", value: "Albanian", docs: 1}
[ more ]	
"Germany"	{facet: "language", value: "English", docs: 5}
"Germany"	{facet: "language", value: "German", docs: 28}
[ more ]	
"Germany"	{facet: "_total_", value: "_total_", docs: [d91, d92, ...]}
"Germany"	{facet: "language", value: "English", docs: 170}
"Germany"	{facet: "language", value: "French", docs: 44}
"Germany"	{facet: "language", value: "German", docs: 9987}
"Germany"	{facet: "language", value: "German Sign Language", docs: 10}
[ more ]	
"France"	{facet: "language", value: "French", docs: 107}
[ more ]	

A subsequent reduction step is required to aggregate all keys with identical facet and value attributes, and to group together all aggregates with identical keys. The code for the `reduce` function is given in Fig. 7.

The table’s values are processed one by one, and hashed into an Erlang dictionary structure using the pair of values for `Facet` and `Value` as dictionary key. If the current facet has the value `_total_`, then aggregation is defined in terms of

<sup>5</sup> That is, data curation is performed on the indices rather than the original documents.





```

comb_4(L) ->
  case length(L) < 4 of true -> "please supply lists with length >= 4" ;
    _ -> [ {A,B,C,D,Z} || A <- L,
              B <- L--[A],
              A < B,
              C <- L--[A,B],
              B < C,
              D <- L--[A,B,C],
              C < D,
              Z <- [L--[A,B,C,D]] ]
  end.

```

**Fig. 8.** Combinators in Erlang

This specification leads to the generation of 121 views, with each view having between 5000 and 12000 bytes of Erlang code.

For the implementation of faceted search, not all possible combinations of set intersections are necessary. The document sets resulting from first selecting facet  $F_1$  and then selecting facet  $F_2$  are identical to those when  $F_2$  is selected first and then  $F_1$ . The computation of all necessary intersections can be done in a very declarative manner using Erlang combinators, see Fig. 8. It computes all necessary intersections for the case in which 4 out of all facets were selected. The predicate `comb_4` gets a list of all facets  $L$ , and returns a list of 5-tuples  $A$ ,  $B$ ,  $C$ ,  $D$ ,  $Z$  such that each of  $A$ ,  $B$ ,  $C$ , and  $D$  stems from  $L$ , are all different and lexically ordered, and where  $Z$  is the list of facets with  $A$ ,  $B$ ,  $C$ , and  $D$  removed from  $L$ . Their output informs the generation of code similar to the one given in Fig. 6. Similar predicates have been written for `comb_1` to `comb_3` to inform the generation of views for cases where users selected between one and three facets.

## 4.5 Web Interface

We have implemented the faceted browser's GUI in JavaScript using the JQuery library ([jquery.com](http://jquery.com)). Facets are represented as tables, where each table column shows a given facet's value, together with the number of associated documents. Clicking on a column causes a HTTP request, which encodes the selection, to be sent to the REST-based CouchDB database, the result of which is used to update all tables. The history of facet selections is maintained in terms of URL parameters. Clicking on "Germany" in the facet table "country", for instance, translates into a CouchDB query

```
/mpi_mgt/_design/country/_view/country?key=%22Germany%22&reduce=true
```

yielding a JSON data structure

```

{"rows": [
  {"key": "Germany", "
    value": [
      {"facet": "modality", "value": "Unspecified", "docs": 140},
      {"facet": "modality", "value": "Speech/gestures", "docs": 230},
      {"facet": "language", "value": "German Sign Language", "docs": 433},
      {"facet": "genre", "value": "Secondary document", "docs": 3},
      {"facet": "genre", "value": "Movie", "docs": 458},
      {"facet": "_total_", "value": "_total_",
        "docs": ["oai:www.mpi.nl:MPI100",
          "oai:www.mpi.nl:MPI1002978"...]}
    ]
  }
  [...]
]}]}

```

NALIDA Faceted Browsing

http://www.sfs.uni-tuebingen.de/nalida/katalog/app/nalida/\_design/nalida/fa

Zentrum für Nachhaltigkeit linguistischer Daten:  
Suche nach Ressourcen

Ein Faceted Browser mit bedingten und unbedingten Facetten

Facet: modality (6)

modality	Occurrences
Unspecified	13
Signs	76
verbal and non-verbal interaction	763
Writing	93
Speech	2671
Pointing gestures	448

Facet: language (6)

language	Occurrences
German Sign Language	10
Swedish Sign Language	15
French	93
British Sign Language	23
Dutch Sign Language	28
German	3446

Facet: country (5)

country	Occurrences
Germany	3450
Netherlands	21
France	93
Sweden	8

resourceclass: Resource

Documents (3618)

documents

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1003709

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1006093

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1047800

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1048528

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1048884

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053705

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053706

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053707

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053708

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053709

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053710

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053711

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053712

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053713

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053714

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053715

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053716

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053717

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053718

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053719

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053720

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053769

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053770

IMDitoOAlbridge\_oai:www.mpi.nl:MPI1053771

Fig. 9. User Interface of the Web Application

which is then processed on the client side to fill the respective tables. The `_total_` facet is entered into the GUI's document table to display all documents identified with a facet selection. We have also provided a full-text search capable of performing its duties on the entire document corpus, or the document set obtained by prior facet selection. The faceted browser is publicly available at [http://www.sfs.uni-tuebingen.de/nalida/katalog/app/nalida/\\_design/nalida/](http://www.sfs.uni-tuebingen.de/nalida/katalog/app/nalida/_design/nalida/).

## 5 Evaluation and Future Work

The computational complexity of faceted browsing stems from the large number of set interactions required for the computation of views. The user of the faceted browser does not need to experience any waiting though, as CouchDB views can be calculated off-line so that every possible user query can be pre-computed and “cached”. The cost of off-line computation, however, is high in the presence of large document sets combined with many facet-value pairs. In the remaining part of the paper, we sketch some of the optimizations to reduce this cost.

*Views for Document Indexing.* The views for document indexing are automatically generated from a facet specification using JavaScript. The resulting map and reduce functions are in JavaScript too, CouchDB's default view language. The computation of the view “organisation”, for instance, takes nevertheless approximately 25 minutes on a set of 86k documents, using a Macbook Pro laptop with a 2.4Ghz Prozessor with 2GBs of memory. Given that this is a one-time payoff, no effort has been made yet to increase the speed of view computation. Moreover, small changes in the document database will have only a small impact on view recomputation at the document indexing level.

*Views for Faceted Search.* The computation of views to inform faceted search is computationally expensive. Prior efforts to implement and run map functions in JavaScript were unsuccessful and soon abandoned. First experiments with Erlang, however, lead to encouraging results, both in terms of memory and processor usage. The following data was computed with the following configuration: (i) each Erlang view was stored in a separate CouchDB design document to better control view computation;<sup>6</sup> (ii) as the map-reduce framework lends itself to parallelization, and given the high number of map-reduce functions to be run on large document sets, we moved view computation to a Ubuntu-driven 24-core machine with 96GB of memory.<sup>7</sup>

For evaluation purposes, we harvested approximately 86.000 metadata documents on language resources from the OAI-PMH server of the Max Planck Institute for Psycholinguistics and ingested them into CouchDB. We have used

<sup>6</sup> A view is computed when it is accessed for the first time, or when a view in the same design document is computed.

<sup>7</sup> Precomputed views can be copied from powerful multi-core machine to the webserver hosting the faceted browser once their databases have been synchronised. With views precomputed, a modest-sized web server is thus sufficient to host the faceted browser.

five unconditional facets “language” (371), “country” (67), “organisation” (39), “modality” (32), and “genre” (50), with the number of facet values given in parentheses. We also indicate the largest document set for each facet: “modality” = “speech” (59463); “language” = “Dutch” (18345); “country” = “Germany” (16178); “organisation” = “Max Planck Institute for Psycholinguistics” (16568), and “genre” = “Discourse” (33676). The automatic generation of faceted search views from the facet specification yielded 31 different map-reduce pairs.

The generation of the views “language”, “country”, “organisation”, “modality”, and “genre” takes altogether less than one minute (using 5 cpus). The generation of the ten views capturing the case where users selected two facets (“country”:”genre”, “country”:”language”, ... “modality”:”organisation”) was also computed in less than 1 minute (using 10 cpus). The computation of the ten necessary views capturing the case where users selected three facets cost altogether less than 7.5 minutes. The most expensive views are those capturing the 5 necessary views that cover the case where users selected four facets. The view “country”:”genre”:”language”:”modality”, for instance, takes more than 2 hours to compute.

*Future Work.* Given evaluation results, there are a number of improvements that we wish to make for the NaLiDa faceted browser. In the live-system, we have introduced indexing documents for each of the metadata providers to exploit the incremental nature of map-reduce. Thus, an update from one data provider only requires a limited view recomputation. Nevertheless, given that some data providers such as the MPI provide 10.000s of documents, we would like to investigate ways to optimise the way index documents for faceted search are managed. Any addition of new metadata descriptions, for instance, could be reflected by a new index document, so that incremental updates are indeed limited to document additions. To account for the cases of metadata modification and deletion, we would like to investigate the introduction of MODIFY and DELETE lists that a revised map-reduce combination would need to consider.

## 6 Related Work and Conclusion

*Flamenco.* The Flamenco toolkit offers a web-based interface to give faceted access to large data collections [6]. It expects developers to prepare a given collection by assigning each of its item to at least one term (say “German”) from one or more facets (say “language”). To assign all collection data into faceted categories, the following tab-delimited files must be provided: the file **facets.tsv** listing all facets, the file **attrs.tsv** listing all attributes of a given item, and the file **items.tsv** listing each collection item (following the attribute definition given in **attrs.tsv**) with a unique integer id. For each entry **facet** in **facets.tsv**, a file *facet\_term* and *facet\_map* must be given. The first file lists all terms for the given facet, together with a unique facet term id; and the second file associates item ids to facet term ids. Once a collection of item is indexed in such a way, it can be ingested into the Flamenco relational database

(MySQL, see <http://www.mysql.com>). With the database in place, the Flamenco toolkit generates the faceted browser's default graphical user interface (GUI), which however, is open to customization. The user's selection of a facet term, or combinations thereof, are translated into corresponding MySQL queries to compute all necessary set interactions. The results of executing MySQL queries are cached to avoid their (potentially costly) re-computation.

A faceted search access to language resources has been implemented by the author (see [7] and <http://www.clarin.eu/vlo>) employing the Flamenco toolkit using roughly the data described in evaluation. Using Perl, we translated the 80.000+ XML-based metadata files into the indexing data format required by Flamenco. We have augmented the Perl-based indexing to also take care of data curation. Once all data was prepared, we ingested the data into the Flamenco database, and adapted its GUI to better fit our needs. To cope with the large dataset, we have written a script whose execution generated all possible queries to warm-up the cache before the faceted browser went live.

The data preparation required for Flamenco roughly corresponds to our CouchDB-based document indexing phase described in Sect. 4.2. In our new approach, data curation, however, only happens when the *views* of the indexing phase are converted into the indexing *documents*, see Sect. 4.3. The MySQL queries fired by Flamenco correspond to the views computed in terms of the indexing documents (see Sect. 4.4).

Our new approach using CouchDB and Erlang has four main advantages. First, CouchDB also stores the original metadata documents (with varying schemata) and thus also serves as permanent storage for this data (see Sect. 4.1). Second, the use of conditional facets contributes to usability as only relevant facets are shown, guiding users navigation; moreover, such facets need only be computed in subsets whose documents are indexed against facet terms the conditional facet depends on. Third, index generation accommodates for incremental updates on the metadata sets, supporting regular harvesting without recomputing all indices and views anew. In Flamenco, any change in the data set would require a full phase of reindexing and index ingestion, overwriting all contents of the previous database, together with its cache. Fourth, our facet specification offers a more declarative view from which we generate procedures for indexing. Index generation is thus taken to a higher level, which make it easier to experiment with different facet configurations, or to adapt an existing one for other datasets. Once a facet specification is changed, however, index generation starts from scratch.

It shows that CouchDB with its native language Erlang is well suited for the development of industrial-strength applications. CouchDB's REST-based interface offers a lean alternative to established software such as the Java-based Apache Tomcat webserver (see <http://tomcat.apache.org/>). We felt Erlang's main limitations being the lack of a full macro package allowing users to write programs to write other programs. A macro capability such as the `defmacro` available in Common Lisp would have made our code generation for faceted search views much easier. In this respect, it is unfortunate that there is no

*strong* support for a Lisp (or Haskell) port to index and query documents in CouchDB. We felt CouchDB's main limitation – when used with Erlang – being the lack of documentation and example code available.

In this paper, we have described a particular application to aggregate heterogeneously structured documents in the domain of linguistic resources and to make them accessible via faceted (and full-text) search. From our description, it should be clear, however, that our approach applies to documents of any domain of discourse or structure as long as their relevant content can be described in terms of JSON (CouchDB's native format).

For our well-defined application context, it was straightforward to give a facet specification as there is only a limited number of information available that is shared across our various metadata sets. In the general case, it would be desirable to detect good facet candidates automatically. The *Castanet algorithm* goes into this direction [8]. It requires the definition of a set of *target terms* to best reflect the topics present in a given document collection. The target terms are then combined with the hypernymy (IS-A) information of the lexical database WordNet (see <http://wordnet.princeton.edu/>) to *both* build facet hierarchies and to assign documents to the facets.

**Acknowledgements.** The NaLiDa project is funded by the German Research Foundation. The data used in the evaluation was harvested from the OAI-PMH server at the Max-Planck Institute for Psycholinguistics in Nijmegen (thanks to Lari Lampen for making the data accessible). The views were computed using a 24-core Linux machine at the University of Tuebingen (thanks to Jochen Saile for making the hardware available). Thanks to Thorsten Trippel for commenting on a prior draft of this article, and to the three anonymous reviewers for their valuable feedback.

## References

1. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide. O'Reilly Media, Sebastopol (2010), <http://guide.couchdb.org>
2. Barkey, R., et al.: Trailblazing through forests of resources in linguistics. In: Digital Humanities. Stanford University, Stanford (2011), [dh2011.stanford.edu/](http://dh2011.stanford.edu/)
3. Cesarini, F., Thompson, S.: Erlang Programming – A Concurrent Approach to Software Development. O'Reilly Media, Sebastopol (2009)
4. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI 2004: Sixth Symposium on Operating System Design and Implementation (December 2004)
5. English, J., Hearst, M., Sinha, R., Swearingen, K., Ping, Y.: Flexible search and navigation using faceted metadata (January 2002) (unpublished manuscript)
6. Hearst, M.A.: Design recommendations for hierarchical faceted search interfaces. In: ACM SIGIR Workshop on Faceted Search (2006)

7. Ringersma, J., Zinn, C., Koenig, A.: Eureka! – User friendly access to the MPI linguistic data archive. *SDV – Sprache und Datenverarbeitung/International Journal for Language Data Processing* 34(1) (2010); *Usability Aspects of Hypermedia Systems*, Cölfen, H., Schmitz, H.-C., Schmitz, U., Schröder B. (eds.), ISBN 978-3-940251-98-5
8. Stoica, E., Hearst, M.A., Richardson, M.: Automating creation of hierarchical faceted metadata structures. In: Sidner, C.L., Schultz, T., Stone, M., Zhai, C. (eds.) *HLT-NAACL*, pp. 244–251. The Association for Computational Linguistics (2007)
9. Wikipedia. Representational state transfer — Wikipedia, the free encyclopedia(2011), [http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer) (accessed May 06, 2011)



# Logic Java: Combining Object-Oriented and Logic Programming

Tim A. Majchrzak and Herbert Kuchen

Department of Information Systems  
University of Münster  
Münster, Germany  
`{kuchen,tima}@ercis.de`

**Abstract.** We have developed the programming language Logic Java which smoothly integrates the object-oriented language Java and logic programming concepts such as logic variables, constraint solving, and backtracking. It combines the advantages of object-orientation such as easy maintainability and adaptability due to inheritance and encapsulation of structure and behavior with the advantages of logic languages such as suitability for search problems. Java annotations and a symbolic Java virtual machine are used to handle the logic programming concepts. In contrast to previous approaches to integrate object-oriented and logic programming, we preserve the syntax of Java. Our language is not split into two distinguishable parts but as closely integrated as possible. Besides the design and implementation of Logic Java, providing a suitable interface between conventional and logic computations is the main contribution of this paper. A killer application, which can hardly be implemented more elegantly in any other language, is the tool Muggl which systematically generates glass-box test cases for Java programs. Applications requiring a substantial amount of search are also well suited.

## 1 Introduction

Object-oriented programming is the dominating programming paradigm. Concepts such as inheritance and encapsulation of structures and behavior [20] provide advantages w.r.t. maintainability and adaptability [33]. Although all application domains can be handled in principle, there are other programming paradigms which are better suited for specific application areas. For example (constraint) logic languages such as *Prolog* [42] are well-suited for search problems due to their built-in search mechanism [36]. Even though declarative paradigms are seldom used in business contexts, there are exceptions. The functional language *Erlang* [3] is successfully used in the telecommunication industry [32,43]; OCaml [35] is applied in the financial services industry [11]. This observation encourages the development of new problem-adequate languages.

We have developed a novel approach called *Logic Java* which smoothly combines the object-oriented language Java [4] with logic programming concepts such as logic variables, constraint solving, and backtracking. It preserves the

syntax of *Java* and uses Java annotations to add the mentioned concepts. Java compilers can still be used without modification. However, we replace the usual Java virtual machine (JVM) [19] by a symbolic one. This symbolic Java virtual machine (SJVM) provides the usual components of a JVM *and* additional features which are known from virtual machines for logic programming languages such as the *Warren Abstract Machine* (WAM) [1], namely logic variables, choice-points, a trail, and a backtracking mechanism. If logic features are not used, the SJVM behaves just as the JVM and causes little performance overhead.

Logic computations can be nested into conventional Java computations by using a corresponding Java annotation (see Sect. 3). If a computation, which is not using logic variables, is nested into a logic computation, it will just behave as a usual Java computation. There is no need to annotate it as *conventional*. Nesting logic computations into logic computations does not change the evaluation mode. The outermost computation is always a conventional one.

Besides the design and implementation of Logic Java, the interface between both types of computations is one of the main contributions of this paper. We explain it with several examples that can be found in the course of this paper.

We assume the reader to be roughly familiar with Java [4] and logic programming (LP). Logic languages such as Prolog provide so called *logic variables*, which are initially *unbound* and then *bound* to some *constructor terms* during a computation [2]. Such a binding happens if an argument of a *predicate* call is unified with the left-hand side of a Prolog rule. Unifying two terms will cause the occurring logic variables to be bound to terms in such way that both terms become identical. If e.g. the so-called *goal* `nat(X)` is evaluated in the context of the program in Listing 1.1, the system will apply the first rule for the predicate `nat` and bind the logic variable `X` to a term only consisting of the constant `zero` and the goal will succeed since the right-hand side of the rule is empty and hence no further computations are necessary. If the user wants another *solution*, the system will *backtrack*, apply the second rule for `nat`, and bind `X` to the term `suc(Y)`. Since the right-hand side of the second rule is not empty, the computation continues and the subgoal `nat(Y)` is solved e.g. by using the first rule and by binding `Y` to `zero`. Thus, the new solution will bind `X` to the term `suc(zero)`.

**Listing 1.1.** A very simple Prolog program

```
nat(zero).
nat(suc(Y)) :- nat(Y).
```

Our paper is structured as follows. Details of the language design are explained in Sect. 3. Before, we give a detailed overview of related work in Sect. 2. Section 4 describes the implementation of Logic Java based on a SJVM. Section 5 contains a discussion of strengths and weaknesses and remarks concerning limitations of our approach. In Sect. 6 we conclude and point out future work.

## 2 Related Work

There is a plethora of work on the combination of programming languages and on multi-paradigmatic approaches. A much cited paper on the family of concurrent

logic programming languages [38] suggests that research reached its climax at the end of the 1980s. It also cites a lot of lesser known approaches not further discussed here. The high number of approaches that address the idea from many different directions underline the importance of the topic.

The following approaches have been developed from a viewpoint of declarative programming. They include *syntactic sugar* to embed object-orientation (OO) features into declarative languages. The power of these languages is preserved. This makes them interesting for declarative programmers but using the OO extensions is not necessarily convenient. Thus, these languages will hardly be given attention by OO programmers. *Oz* is a lazy constraint language with concurrency that offers some object-oriented features [41]. Many multi-paradigmatic languages are based on Haskell or Curry [16], which by itself is multi-paradigmatic and combines functional and logic programming. There are extensions for an object-oriented design [17]. Special approaches incorporate further paradigms leading to e.g. constraint functional logic programming [12].

Prolog has frequently been combined with object-orientation. A classic paper by MCCABE [25] presents a new language and case studies of an OO language on top of Prolog. A 1983 approach discusses object-oriented programming in concurrent Prolog [39]. *Visual Prolog* (formerly known as *Turbo Prolog* [15]) offers an OO extension for Prolog [37]. Its main purpose is the design of artificial intelligence applications, though. *Logtalk* [30] and *Prolog++* [29] are two well known approaches of adding OO features to Prolog. Regardless of the integration of object-orientation, their syntax is similar to that of pure Prolog.

Declarative meta languages such as *SOUL* [26] only roughly relate to our approach. Despite offering object-oriented functionality, embedding logic programming into an OO language is not their main purpose. For more details on the application of *SOUL* e.g. check [10]. Not only general approaches exist. Many authors address niche problems. E.g., there is a fuzzy object-oriented logic programming system [57], an object-oriented logic language for modular system specification [28] and an object-oriented logic programming environment for modeling [34]. Even a US patent (# 4.989.132) of a “tool [...] which integrates an object-oriented programming language system, a logic programming language system, and a database” exists. Other approaches address special contexts such as distributed computing, e.g. *ObjVProlog-D* [24]. Due to their special nature, our work does not compete with any of these approaches.

All approaches so far discussed combine more than one paradigm. Yet their concepts differ from our ideas. Most notably, in almost all cases a logic language is extended with object-oriented functionality or the extension of an OO language significantly changes it. Both issues hinder a widespread reception of the languages. There is one recent approach which has similar concepts as ours. *CIMADAMORE* and *VIROLI* combine Java with Prolog [8,9]. Despite their use of generics and annotation, their work has a different focus. It allows Prolog code to be integrated into Java and enables an exchange of data between Java and Prolog. While we embed logic programming into a Java virtual machine (VM), *CIMADAMORE* and *VIROLI* start with an existing Prolog engine.

## 3 Design of Logic Java

### 3.1 General Principles

Before presenting Logic Java, we would like to state its design principles. Firstly, the language should be *easy to learn for Java users*. We reach this by preserving the Java syntax. This also ensures that we trivially reach the second aim: the language should be as *homogeneous* as possible. In particular, there should be no different syntax for logic and OO computations. Thirdly, there should be *no performance penalty* for conventional Java computations. We guarantee this by using a symbolic Java virtual machine which behaves as the conventional Java virtual machine if no logic computations are required. Fourthly, *several search strategies* should be supported. For efficiency, Prolog just provides the (incomplete) *depth-first search* strategy. This causes problems when writing Prolog programs. Essentially, the declarative character of the language is lost since the programmer has to avoid infinite computations. Our implementation provides the complete search strategy *iterative deepening* in addition to depth-first search. Other strategies such as breadth-first search can be added since the strategy is a modular and modifiable part of the implementation. And fifthly, *we do not mean to change Java* in a way that would require programmers to change the way they use it. Rather, we want to augment it with new constructs. Consequently, it can be perfectly used as it always was – or, if problems demand it, in its extended version with the additional power of another paradigm. We fully adhere to the Java language specification [14].

We now describe the design of the language. Logic programming concepts, namely *logic variables*, *unification*, and *backtracking*, will have to be provided or appropriately replaced when integrating Java and logic programming. There are (at least) three feasible ways for introducing logic variables:

- Providing a generic type `LogicVariable<T>` which marks variables of type `T` as logic variables.
- Annotating variables as logic variables.
- Introducing a default initialization as logic variables depending e.g. on a specific naming scheme (as e.g. in Fortran [27]).

Using generic types (*generics* [31]) is a strategy often sought. However, a wrapper class causes overhead at runtime due to the costs of object generation and (automated) (un-)boxing. A default initialization depending on naming is error prone and inflexible. Using annotations does not only provide full flexibility but also offers the best runtime characteristics. In particular, it is possible to annotate primitive types. We therefore introduce the annotation `@LogicVariable` which can be used on fields (class members) and local variables of methods.

Unification is a special case of constraint solving and many practical Prolog programs do not only rely on the rather simple constructor term unification but integrate domain specific constraint solvers which lead to a smaller search space and more efficient programs. Logic Java provides just constraint solving and no non-trivial unification for parameter passing. Unification can however be simulated using equality constraints.

The SJVM adds new constraints to the constraint store when it processes a conditional jump instruction such as `if_icmpeq` in Java bytecode. In case of `if_icmpeq` this is an equality constraint relating the two topmost entries on the stack. Other conditional jump instructions produce disequality or inequality constraints. Moreover, the SJVM will create a *choice point*. After finishing the first alternative of the computation (successfully or not), the SJVM will backtrack and replace the mentioned constraint in the constraint store by its negation.

If a method shall be evaluated as a logic computation using logic variables, constraint solving and backtracking, we annotate it with `@Search`. This annotation offers optional parameters for configuration. `strategy` can be set to change the search strategy used when executing logically. At the moment, depth first search (`SearchStrategy.DEPTH_FIRST`) and iterative deepening depth first (`SearchStrategy.ITERATIVE_DEEPENING`) are supported. The latter forces *backtracking* if no solution has been found when reaching a specified depth of search. The parameter `deepeningStartingDepth` can be set to a positive integer value. By specifying a `deepeningIncrement`, search will start over with a maximum depth which is set to the sum of the two parameters.

### 3.2 Introductory Examples

Let us illustrate Logic Java and its concepts using an example. The method `smm()` finds a solution for a classical problem solved by logic programming: `SEND + MORE = MONEY` where each character has to be replaced by a different digit (Listing 1.2). `SendMoreMoney` is a normal Java class. It has eight fields annotated with `@LogicVariable` to represent the characters. `smm()` is annotated with `@Search` to enable logic processing. The search strategy used is depth first; the parameter could have been omitted since this is the default search strategy. Please note that `allDifferent(int[])` does *not* need to be annotated since the nested call passes logic variables to it.

The outer condition of the `if` statement represents the main problem:  $((1000s + 100e + 10n + d) + (1000m + 100o + 10r + e)) = (10000m + 1000o + 100n + 10e + y)$ . The inner condition of the `if` statement ensures that variables assume pairwise different values. If any condition is not satisfied, an `EmptySolution` is generated. This signals that the currently considered branch of the computation could not provide a solution. If conditions are satisfied, i.e. the puzzle is solved, the values of the eight variables are (in this case) stored as an array and saved as a `Solution`. The `Solutions` container is used and either takes an instance of `Solution` or `EmptySolution` as an argument.

The predefined class `Solutions` provides the interface between logic and conventional Java computation. Note the quantity mismatch at this point. From a Java point of view, the constructor `Solutions` gets just one – possibly empty – solution as argument. The SJVM treats the type `Solutions` specifically. Rather than returning a single solution directly, it collects all solutions of a logic computation and returns them after all branches of the logic computation are finished.<sup>1</sup> Duplicate empty solutions are removed. If there is a non empty solution, empty

<sup>1</sup> Infinite computation can be avoided by using the iterative deepening search strategy.

solutions are removed. In a simple case like the present one and also for the majority of applications, a solution just consists of a value without references to logic variables. Here, this value is an array of integers. Such a solution can be fetched by the enclosing computation using the `getSolution()` method. The `main(String... args)` method shows how this can be done. Later, we will describe the general case, which is a bit more complicated to handle.

**Listing 1.2.** Send more money in Logic Java

```
public class SendMoreMoney {
    @LogicVariable
    protected int e, d, m, n, o, r, s, y;

    @Search(strategy=SearchStrategy.DEPTH_FIRST)
    public Solutions<Integer[]> smm() {
        if ( (s * 1000 + e * 100 + n * 10 + d)
            + (m * 1000 + o * 100 + r * 10 + e)
            == (m * 10000 + o * 1000 + n * 100 + e * 10 +
                y)) {
            int[] variables = {e, d, m, n, o, r, s, y};
            if (allDifferent(variables)) {
                Integer[] solution = {e, d, m, n, o, r, s, y};
                return new Solutions(new
                    Solution<Integer[]>(solution));
            }
        }
        return new Solutions(new EmptySolution());
    }

    public boolean allDifferent(int[] a) {
        for (int i = 0; i < a.length; i++) {
            for (int j = i + 1; j < a.length; j++) {
                if (a[i] == a[j]) return false;
            }
        }
        return true;
    }

    public static void main(String... args) {
        SendMoreMoney sendM = new SendMoreMoney();
        Solutions<Integer[]> solutions = sendM.smm();

        for (Solution<Integer[]> solution : solutions) {
            Integer[] values = solution.getSolution();
            for (int value : values)
                System.out.println(value);
        }
    }
}
```

The `main(String...)` method executes `smm()`, gets the solutions and iterates over them. `Solutions` implements the `Iterator` interface which is known to be very helpful [13]. It can be used for handling the solutions one by one. In this example, we simply write the values of the variables to standard output. We do not need to explicitly state that the values of variables should be single digits since the first solution returned will be the simplest one.

Note that a pure Java implementation of the send-more-money example would need to program the search explicitly, e.g. by using 8 nested loops. This would be a bit clumsy but manageable. The 8 nested loops would essentially correspond to depth-first search. Since the search space is finite, the solution would be found eventually. In situations where depth-first search runs into an infinite computation, a complete search strategy such as iterative deepening would have to be implemented explicitly. Then, a pure Java program would be extremely more complex than a Logic Java program.

Consider a Logic Java program which computes all solutions of Fermat's problem (Listing 1.3). It finds suitable natural numbers  $a, b, c, n$  such that  $a^n + b^n = c^n$ . The trivial implementation of method `power(int, int)` is omitted. We encourage the reader to try to implement a corresponding pure Java program.

**Listing 1.3.** Fermat's Last Theorem in Logic Java

```
public class Fermat {
    @LogicVariable
    protected int a, b, c, n;

    @Search(strategy=SearchStrategy.ITERATIVE_DEEPENING,
            deepeningIncrement=5)
    public Solutions<Integer[]> fermat() {
        if (power(a,n) + power(b,n) == power(c,n)) {
            Integer[] solution = { a, b, c, n };
            return new Solutions<Integer[]>((
                new Solution<Integer[]>(solution) ));
        } else {
            return new Solutions<Integer[]>((new
                EmptySolution()));
        }
    }
}
```

In the above example, all logic variables are bound to values by equality constraints on integers. As explained, such a solution can be fetched by the enclosing computation using the `getSolution()` method. In general, a solution consists of a resulting value and a set of constraints which have been accumulated when producing the result. The result can be a logic variable (represented by a corresponding predefined type) or it may contain (possibly indirect) references to logic variables. A constraint is also represented by an object of a corresponding predefined class and it may (possibly indirectly) refer to objects representing logic variables. If the user wants to process such a solution, appropriate

predefined methods of class `Solutions` and other predefined classes can be used in order to extract the required information. A complete description of the predefined classes for internally representing solutions, logic variables, and constraints is out of scope of this paper. We briefly sketch some possibilities only.

Besides `getSolution()` the class `Solution` offers additional methods for extracting the result and constraints of a solution and for working with them. The following list shows the most important methods. Assuming that a solution consists of a result  $r$  and a set  $s$  of constraints, both may contain logic variables.

`void addConstraint(Constraint)` adds a constraint to  $s$ . When adding it, the solver is invoked and the changed constraint system processed.

`boolean isSatisfiable()` checks whether  $s$  is still satisfiable.

`T findExampleResult()` finds an arbitrary example result of type  $T$  which is obtained by instantiating all logic variables occurring in  $r$  by values which are free of logic variables (so called *ground* values). The chosen values have to correspond to  $s$ . E.g. if  $r$  consists of  $X + 5$  (where  $X$  is a logic variable) and  $s$  is the set  $\{X \leq 3, X > 1\}$ , possibly delivered results can be 7 and 8, respectively. One of them will be chosen randomly.

`isGround()` checks whether  $r$  does not contain logic variables.

#### Listing 1.4. `inInterval` in Logic Java

```
public class Value {
    @LogicVariable
    protected double x;

    @Search(strategy=SearchStrategy.ITERATIVE_DEEPENING,
            deepeningIncrement=5)
    public Solutions<Double> inInterval(double x1, double
        x2) {
        if (x1 <= x && x <= x2)
            return new Solutions<Double>(new
                Solution<Double>(x));
        else
            return new Solutions<Double>(new
                EmptySolution());
    }

    public static void main(String... args) {
        Solutions<Double> solutions
            = (new Value()).inInterval(3.0, 5.0);
        for (Solution<Double> sol : solutions)
            if (sol.isGround())
                System.out.println(sol.getSolution());
            else
                System.out.println(
                    sol.findExampleResult().getSolution());
    }
}
```



Listing 1.4 demonstrates the exemplary usage of some of the methods provided with Logic Java. `inInterval(double, double)` checks whether the value of the variable  $x$  is contained in the interval determined by both parameters. Note that in contrast to many other integration approaches Logic Java seamlessly works with arbitrary basic values such as doubles. Any primitive type of Java can be used: integers (byte, char, short, int, long), floating point numbers (float, double), and boolean values (boolean). The call of the method inside of method `main(String... args)` returns a `Solutions` object containing one solution consisting of the unbound logic variable  $x$  as result and the set of constraints  $\{x \geq 3.0, x \leq 5.0\}$ . Using `findExampleResult()`, we choose one possible value for  $x$  which corresponds to the constraints, e.g. 4.2.

## 4 Implementation

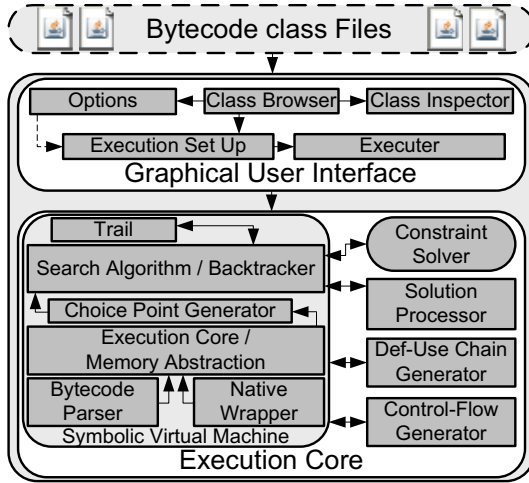
The core of the implementation of Logic Java is the symbolic Java virtual machine. It was adapted from a previous project [22,23]. Using *choice points* and *backtracking*, the SJVM processes all possible paths through a considered Java program read from the bytecode in a `class` file. Choices, e.g. conditional jumps or switching instructions, lead to the generation of *choice points*. For each such instruction, a *constraint* is generated describing the condition under which the considered branch can be entered. This constraint is added to the constraint store. The set of constraints encountered so far is processed by the built-in *constraint solver*. If the current set of constraints is not satisfiable, the considered branch of the computation is abandoned and backtracking occurs.

The architecture of the SJVM is depicted in Fig. 1. The SJVM consist of a conventional JVM which has been extended by features known from virtual machines of logic programming languages, namely logic variables (and corresponding data structures to handle their computation), choice-points, a trail, and a backtracking mechanism. Moreover, it has to manage the search strategy (depth-first search or iterative deepening). An in-depth discussion of these components is out of scope but given in [22].

Our solver has been specifically designed to process constraints generated while executing Java bytecode. During execution, linear and non-linear constraints are encountered. Whilst it is of course possible to program a method to calculate e.g. the logarithm in Java, on a bytecode level only simple instructions are used<sup>2</sup>. This includes basic arithmetic operations (addition, subtraction, multiplication, division, remainder), logic operations (and, or, xor) and conditional jumps. Bit operations occurring in constraints are simulated by arithmetic operations. Moreover, constraints are transformed into certain normal forms, namely equations, disequations, and inequations of polynomials. To be able to handle different types of constraints efficiently, multiple solvers have been implemented including a *simplex* solver for linear equations, a *Fourier-Motzkin* solver for

---

<sup>2</sup> There are (very) complex instructions in Java bytecode such as those for method invocation. However, they are handled the same way in the JVM and SJVM.



**Fig. 1.** Architecture of the Symbolic Java Virtual Machine and auxiliary components

linear inequations and a *bisection* solver for non-linear constraints. More details on transformations and on the solver are given in a distinct article [18].

Fig. 2 shows a small part of the bytecode generated for an implementation of *send-more-money* (cf. Listing 1.2). Numbers in the first column are instruction offsets rather than line numbers. The second column shows the Bytecode instruction and additional bytes, and the third column shows affected variables or source code statements. The operation is briefly explained in the fourth column.

When the instruction at offset 100 is reached, the outer *if* has been executed successfully and the branch where its condition is met is processed. Thus, the constraint on the constraint store before continuing execution is  $((1000s + 100e + 10n + d) + (1000m + 100o + 10r + e)) == (10000m + 1000o + 100n + 10e + y)$ . Processing the inner *if* begins by pushing the logic variables *e* and *d* onto the stack. Field access is fully qualified in Java bytecode i.e. first the object reference (*this*) to the applicable class is loaded onto the stack and then its field is accessed. Eventually, the instruction at offset 108 compares the two values. Please note that the comparison in Java bytecode is negated; while we want to check for inequality, the instruction checks equality. If  $e = d$  execution jumps to the instruction following the successful return from execution (offset 480:  $97 + ((1 < 8) | 127)$ ). If any conditions from the outer or inner *if* are not met, objects for `EmptySolution` and `Solutions` are created and returned. However, if  $e \neq d$  execution continues with the next step (offset 111). Value *e* is loaded and pushed onto the stack again since the next step is its comparison with *m*.

When executing `if_icmpeq`, the constraint  $e = d$  is added to the constraint store. Immediately, the solver checks the constraint system for satisfiability. If it is not satisfiable, execution is not continued and the constraint is removed. Otherwise, execution continues until it is finished or another unsatisfiable constraint is met. In both cases, backtracking is started. Since the instruction offers

off	bytecode	Java	operation
...			
100	<b>aload_0</b>	<i>this</i>	Load <i>this</i> onto the stack.
101	<b>getfield</b> 0 119	<i>this.e</i>	Get value from field <i>e</i> and push it.
104	<b>aload_0</b>	<i>this</i>	Load <i>this</i> onto the stack.
105	<b>getfield</b> 0 123	<i>this.d</i>	Get value from field <i>d</i> and push it.
108	<b>if_icmpeq</b> 1 116	<i>if(e == d)</i>	Generate constraint $e == d$ and a corresponding choice point.
111	<b>aload_0</b>	<i>this</i>	Load <i>this</i> onto the stack.
102	<b>getfield</b> 0 119	<i>this.e</i>	Get value from field <i>e</i> and push it.
...			

**Fig. 2.** First Bytecode example

a second alternative for  $e \neq d$ , this constraint is put onto the constraint store. Again, satisfiability is checked and execution then continues. If all conditions are satisfied, the encountered solution is added to the previously encountered ones. Possibly existing empty solutions are removed. If all solutions have been collected, a new instance of **Solutions** is generated and returned (see Fig. 3).

Additional aspects besides implementing the SJVM had to be considered. First, a new package with support for the annotations has been created that allows to mark logic variables and methods that should be executed in the logic computation mode. Second, new classes have been introduced to store solutions and make them accessible.

The SJVM ensures that logic variables are initialized correctly. If an object is instantiated (by the **new** statement of Java), the JVM generates an internal representation of an object reference (*objectref*). It then checks whether fields of the underlying class have been marked with **@LogicVariable**. If so, it initializes the fields to logic variables. Otherwise, they simply take default values and are used as constants. The same applies to local variables of a method that are annotated **@LogicVariable**. They are initialized when the method is invoked and a so called *frame* is generated.

The annotation of logic variables works fine for member variables. Unfortunately, the annotation of local variables of a method is not reflected in class files even if **Target** is set to **ElementType.LOCAL\_VARIABLE** and **Retention** is set to **RetentionPolicy.CLASS** or **RetentionPolicy.RUNTIME**. The claimed technical reasons for the decision to let the Java compiler ignore annotations of local variables are not convincing. We hope that they will be reflected in class files with the release of Java 7. Up to then, only member variables can be used as logic variables. This restriction is unaesthetic but not serious.

Our approach is robust w.r.t unnecessary usage of the annotations. Nesting methods annotated with **@Search** does no harm; the same applies to annotation of variables that are used in searches. Logic variables passed as parameters remain logic variables regardless of a potential duplicate annotation. They will neither be reset nor would the doubled annotation disable logic processing. Calling methods that take a mixture of constant and logic parameters are handled, too: constant values are simply calculated in the constant way. Our JVM

off	bytecode	Java	operation
...			
408	<b>bi</b> push 8		Push 8 onto the stack.
410	<b>new</b> array 10	<i>new int[8]</i>	Generate a new array with 8 elements and push it onto the stack.
412	<b>dup</b>		Duplicate the topmost stack element i.e. the array.
413	<b>iconst</b> _0		Push 0 onto the stack.
414	<b>aload</b> _0	<i>this</i>	Load <i>this</i> onto the stack.
415	<b>getfield</b> 0 119	<i>this.e</i>	Get value from field <i>e</i> and push it.
418	<b>istore</b>		Save <i>e</i> as the first elements of the array.
...			
470	<b>astore</b> _1		Save the array to local variable 1.
471	<b>new</b> 0 33	<i>new</i>	Create the new object.
474	<b>dup</b>		Duplicate the topmost stack element i.e. the object reference.
475	<b>aload</b> _1		Load the array onto the stack.
476	<b>invokespecial</b> 0 133		Collect all solutions.
479	<b>areturn</b>	<i>return</i>	Return from execution with the topmost element from the stack i.e. the object reference.
...			

**Fig. 3.** Second Bytecode example

adheres to the the specification [19]; the decrease in efficiency between “normal” and logic mode on constant operations is negligible.

## 5 Strength and Limitations

Our logic extension of Java is suited to applications that require a substantial amount of search. In particular, combinatorial problems such as *n-queens*, and *graph coloring* can easily be handled. But Logic Java is not only suited for toy problems. Also practically relevant combinatorial problems such as *crew scheduling problems*, *machine planning* and various forms of allocation and resource planning problems can be handled. However, we cannot easily find optimal solutions since Logic Java does not include an optimization algorithm. However, with help of constraints corresponding to a lower or upper bound of some objective function only solutions of a certain minimal quality will be considered. Logic Java is also well suited for certain games. For example, it can be used in artificial intelligences that determine the moves of computer opponents, in particular if this requires the exploration of large search spaces.

One of the most convincing applications of Logic Java is the test-case generator Muggl [22]. Muggl systematically generates a minimal set of glass-box test cases for Java classes such that predefined code coverage criteria such as control-flow and/or data-flow coverage are met. It calls the method that shall be tested with logic variables as parameters. Each solution of a symbolic execution of the code provides a system of constraints which describes a set of equivalent test

cases (w.r.t. the coverage criterion). Any solution of such a system of constraints can be used as a test case, which tests all equivalent behaviors of the program.

Compared to other approaches which integrate object-oriented and logic programming (see Sect. 2), Logic Java has the advantage that it is very close to Java; in fact it subsumes it. Most practitioners use OO languages for their daily routine. OO languages are taught in most computer science degree programs and they are well understood by almost all scientists that require programming as well. Thus, it should be easier for them to learn Logic Java than a logic programming language which has been extended by object-oriented features.

It often requires a lot of programming or leads to programs that are hard to read if problems similar to our examples are implemented in pure OO languages. At the same time, logic programming is rarely used for practical applications. Using the flexibility and versatility of an OO language and the libraries available for it is especially appealing if logic sub-routines can be used. Implementing logic programs exactly where you need them and without the need to write wrapper classes or to establish links between single software systems is extremely helpful.

Logic Java has strengths beyond the amenities of the general multi-paradigmatic approach. First of all, it is very easy to learn for users that have knowledge in OO programming. Of course, advanced concepts to successfully create logic programs are (very) hard to master. Nevertheless, users of Logic Java can broaden their knowledge as required for the tasks of their choice while starting with nothing but knowledge of Java. Secondly, the sophisticated functions to alter solutions, modify the constraints and get expressions calculated by the solver are very powerful tools. Users with advanced logic programming knowledge will find them useful. Thirdly, domains formerly dominated by logic programming become open for object-oriented programs. In particular, it becomes possible to interchange data between the “two worlds”. And fourthly, annotations offer a very flexible way to implement the logic features.

Our approach also has some limitations. As already mentioned, due to the Java compiler local variables cannot be used as logic variables. Instead, member variables have to be used, until the mentioned problem is fixed, possibly with Java 7. While Logic Java has the basic functionality of Prolog, some features are missing. Most notably, there is no equivalent for the cut operator `!`. However, it is not difficult to simulate it, e.g. by using globally available static class variables. They guide the control flow and the backtracking mechanism.

It is currently not possible to combine logic and concurrent programming in Logic Java. While the JVM offers concurrency, it is not yet implemented for symbolic computations. Combining concurrency with symbolic computation is very tricky, in particular in the presence of backtracking. One could use some of the concepts taken from implementations of or-parallel Prolog versions such as Aurora [21,40]. However, we have not yet done so. At the moment, threads can only be used outside of logic computations and they must not interact with them.

Currently, only our JVM can be used to execute programs written in Logic Java. Since we used annotations to add the logic functionality, an *Annotation*

*Processor* [6] in combination with a library with the backtracking and solving functionality could be used to run Logic Java on any standard compatible [19] JVM.

## 6 Conclusion and Future Work

We have presented the programming language Logic Java, which combines object-oriented and logic programming. Starting from a discussion of related work, we formulated design goals and introduced the concepts of Logic Java. Along with several examples we explained its implementation as well as the interface between logic and conventional computations. Then we identified strengths and limitations of the approach and we named suitable application areas.

Studying previous approaches we observed that they often expand a language or even invent a new one. Most of them require programmers to learn new concepts and to distinguish different syntactic categories for logic and conventional object-oriented computations. We believe that our approach to combine object-oriented and logic programming is smoother than others. We preserve the Java syntax and we use a single execution mechanism for logic and conventional Java computations, namely the Symbolic Java Virtual Machine. Even though we identified some limitations of our approach, there are ways to decrease the negative effects or even circumvent many of them.

A strength of our approach is the integrated constraint solver. Even though it would be possible to work with external constraint solving libraries to solve problems such as *send more money* in Java, using Logic Java is more convenient and more versatile. Programmers hardly have to think about constraint solving and do not have to *program* the solver; constraint solving is neatly integrated into logic computation and done to speed up execution.

We will continue by refining Logic Java. A main aim of our future work will be the extensive experimental evaluation of our approach. More examples have to be implemented and thoroughly tested. Results gained from this will facilitate further improvement of the specification.

## References

1. Ait-Kaci, H.: Warren's abstract machine: a tutorial reconstruction. MIT Press, Cambridge (1991)
2. Apt, K.R.: From logic programming to Prolog. Prentice-Hall, Upper Saddle River (1996)
3. Armstrong, J.: The development of Erlang. In: ICFP 1997: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, pp. 196–203. ACM, New York (1997)
4. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language, 4th edn. Addison-Wesley, London (2005)
5. Baldwin, J., Martin, T., Vargas-Vera, M.: Fril++: object-based extensions to Fril. In: Martin, T., Fontana, F. (eds.) *Logic Progr. and Soft Computing*, pp. 223–238. Research Studies Press, Hertfordshire (1998)

6. Bloch, J.: *Effective Java*, 2nd edn. Prentice Hall, Upper Saddle River (2008)
7. Cao, T.H., Rossiter, J.M., Martin, T.P., Baldwin, J.F.: On the implementation of Fril++ for object-oriented logic programming with uncertainty and fuzziness. *Technologies for Constructing Intelligent Systems: Tools*, 393–406 (2002)
8. Cimadamore, M., Viroli, M.: A Prolog-oriented extension of Java programming based on generics and annotations. In: *Proceedings PPPJ 2007*, pp. 197–202. ACM, New York (2007)
9. Cimadamore, M., Viroli, M.: Integrating Java and Prolog through generic methods and type inference. In: *Proc. SAC 2008*, pp. 198–205. ACM, New York (2008)
10. D'Hondt, M., Gybels, K., Jonckers, V.: Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In: *Proc. of the 2004 ACM SAC*, SAC 2004, pp. 1328–1335. ACM, New York (2004)
11. Eber, J.M.: The financial crisis, a lack of contract specification tools: What can finance learn from programming language design? In: Castagna, G. (ed.) *ESOP 2009*. LNCS, vol. 5502, pp. 205–206. Springer, Heidelberg (2009)
12. Fernandez, A.J., Hortala-Gonzalez, T., Saenz-Perez, F., Del Vado-Virseda, R.: Constraint functional logic programming over finite domains. *Theory and Practice of Logic Programming* 7(5), 537–582 (2007)
13. Gibbons, J., Oliveira, B.: The essence of the iterator pattern. *J. Funct. Program.* 19(3-4), 377–402 (2009)
14. Gosling, J., Joy, B., Steele, G., Bracha, G.: *The Java(TM) Language Specification*, 3rd edn. Addison-Wesley Professional, London (2005)
15. Hankley, W.J.: Feature analysis of turbo prolog. *SIGPLAN Not.* 22, 111–118 (1987)
16. Hanus, M., Kuchen, H., Moreno-Navarro, J.: Curry: A Truly Functional Logic Language. In: *Proceedings ILPS 1995 Workshop on Visions for the Future of Logic Programming*, pp. 95–107 (1995)
17. Kuchen, H.: Implementing an Object Oriented Design in Curry. In: *Proceedings WFLP 2000*, pp. 499–509 (2000)
18. Lembeck, C., Caballero, R., Mueller, R.A., Kuchen, H.: Constraint solving for generating glass-box test cases. In: *Proceedings WFLP 2004*, pp. 19–32 (2004)
19. Lindholm, T., Yellin, F.: *The Java Virtual Machine Specification*, 2nd edn. Prentice-Hall, Englewood Cliffs (1999)
20. Loudon, K.C.: *Programming Languages*. Wadsworth, Belmont (1993)
21. Lusk, E., Butler, R., Disz, T., Olson, R., Overbeek, R., Stevens, R., Warren, D.H., Calderwood, A., Szeredi, P., Haridi, S., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B.: The Aurora or-parallel Prolog system. *New Gen. Comput.* 7(2-3), 243–271 (1990)
22. Majchrzak, T.A., Kuchen, H.: Automated Test Case Generation based on Coverage Analysis. In: *TASE 2009: Proceedings of the 2009 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering*, pp. 259–266. IEEE Computer Society, Los Alamitos (2009)
23. Majchrzak, T.A., Kuchen, H.: Muggl: The Muenster Generator of Glass-box Test Cases. In: Becker, J., Backhaus, K., Grob, H., Hellgrath, B., Hoeren, T., Klein, S., Kuchen, H., Müller-Funk, U., Thonemann, U.W., Vossen, G. (eds.) *Working Papers No. 10*. European Research Center for Information Systems, ERCIS (2011)
24. Malenfant, J., Lapalme, G., Vaucher, J.: ObjVProlog-D: a reflexive object-oriented logic language for distributed computing. In: *Proceedings OOPSLA/ECOOP 1990*, pp. 78–81. ACM, New York (1991)
25. McCabe, F.G.: *Logic and objects*. Prentice-Hall, Upper Saddle River (1992)
26. Mens, K., Michiels, I., Wuyts, R.: Supporting software development through declaratively codified programming patterns. *Expert Syst. Appl.* 23(4), 405–413 (2002)

27. Metcalf, M., Cohen, M.: Fortran 95/2003 Explained, 3rd edn. Oxford University Press, Oxford (2004)
28. Morzenti, A., Pietro, P.S.: An Object-Oriented Logic Language for Modular System Specification. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 39–58. Springer, Heidelberg (1991)
29. Moss, C.: Prolog++: The Power of Object-Oriented and Logic Programming, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1994)
30. Moura, P.: From plain prolog to logtalk objects: Effective code encapsulation and reuse. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 23–23. Springer, Heidelberg (2009)
31. Naftalin, M., Wadler, P.: Java Generics and Collections. O'Reilly Media, Inc., Sebastopol (2006)
32. Nyström, J.H.: Productivity gains with Erlang. In: Proceedings CUFPP 2007. ACM, New York (2007)
33. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992)
34. Page, Jr., T.W.: An object-oriented logic programming environment for modeling. Ph.D. thesis, University of California, Los Angeles (1989)
35. Rémy, D.: Using, understanding, and unraveling the oCaml language from practice to theory and vice versa. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 413–536. Springer, Heidelberg (2002)
36. Salus, P.H.: Functional and Logic Programming Languages. Sams, Indianapolis (1998)
37. Scott, R.: A Guide to Artificial Intelligence with Visual Prolog. Outskirts Press (2010)
38. Shapiro, E.: The family of concurrent logic programming languages. *ACM Computing Surveys* 21(3), 413–510 (1989)
39. Shapiro, E.Y., Takeuchi, A.: Object Oriented Programming in Concurrent Prolog. *New Generation Comput.* 1(1), 25–48 (1983)
40. Szeredi, P.: Solving Optimisation Problems in the Aurora Or-parallel Prolog System. In: ICLP 1991: Pre-Conference Workshop on Parallel Execution of Logic Programs, pp. 39–53. Springer, London (1991)
41. Van Roy, P., Brand, P., Duchier, D., Haridi, S., Schulte, C., Henz, M.: Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming* 3(6), 717–763 (2003)
42. Warren, D.H.D., Pereira, L.M., Pereira, F.: Prolog – the language and its implementation compared with Lisp. In: Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, pp. 109–115. ACM, New York (1977)
43. Wiger, U.: 20 years of industrial functional programming. In: ICFP 2004: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, pp. 162–162. ACM, New York (2004)



# On Proving Termination of Constrained Term Rewrite Systems by Eliminating Edges from Dependency Graphs<sup>\*</sup>

Tsubasa Sakata, Naoki Nishida, and Toshiki Sakabe

Graduate School of Information Science, Nagoya University  
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan  
sakata@sakabe.i.is.nagoya-u.ac.jp,  
{nishida, sakabe}@i.is.nagoya-u.ac.jp

**Abstract.** In this paper, we propose methods for proving termination of constrained term rewriting systems, where constraints are interpreted by built-in semantics given by users, and rewrite rules are assumed to be sound for the interpretation. To this end, we extend the dependency pair framework for proving termination of unconstrained term rewriting systems to constrained term rewriting systems. Moreover, we extend the dependency pair framework so that dependency pair processors take a subgraph of the dependency graph as input and they output a finite set of graphs which can be obtained by eliminating nodes and/or edges from the input graph.

## 1 Introduction

*Constrained (un-)conditional term rewriting systems* are finite sets of constrained (un-)conditional rewrite rules [32,33,8,23,26,5,9,10,6,7,17,30,27], where the constraint parts are evaluated by *built-in semantics* (such as memberships and integer arithmetics) or *equational theories* independent on the rewrite rules, and the condition parts are evaluated by the rewrite rules recursively (and/or partially evaluated by the built-in semantics). Moreover, constrained systems are enriched such as *constrained equational systems* (CESs, for short) [13,14]. This paper deals with *constrained unconditional term rewriting systems* (constrained TRSs, for short) in [6,7,17,30,27], that are based on ones of the general formulations.

Recently, inductive theorem proving methods for constrained TRSs are investigated [5,11,6,12,7,17,30]. Termination of constrained TRSs is one of the most important properties in such methods and thus methods for proving termination of constrained TRSs are expected to be developed. Such methods are also useful in proving innermost termination of unconstrained term rewriting systems (TRSs, for short) since termination of the constrained version of a TRS implies innermost termination of the TRS [4].

---

<sup>\*</sup> This work is partially supported by MEXT KAKENHI #20300010 and # 21700011.

As a termination proof technique for TRSs both termination criterion based on *dependency pairs* and the *dependency pair framework* (the DP framework, for short) are well studied [2,21,19,18,22,15,16], and the framework has been extended to CESs [13,14]. To prove termination by using this framework, we only prove that there exists no infinite *dependency chain* that is a sequence of dependency pairs. In particular, a technique in [20] is useful in proving termination of TRSs via polynomial interpretations over integers. In the technique, the DP processor based on polynomial interpretations over integers eliminates two kinds of dependency pairs: those ordered by  $>$  over integers, and those guaranteeing the existence of a lower bound of integer strictly-decreasing sequences obtained from dependency chains. For each dependency pair in an input set, the processor refers *all* the dependency pairs that possibly follow after the focusing dependency pair in order to use additional information that is useful in ordering the focusing dependency pair or in detecting a lower bound. This technique seems useful in proving termination of constrained TRSs, while the reference to the accident pairs is not implemented in [13,14]. For a dependency pair, however, the processor does not succeed in detecting a lower bound for every accident pair, while the processor detects a lower bound for some of the accident pairs. In such a case, it does not matter to ignore the *paths* from the focusing dependency pair to the accident pairs with a lower bound, i.e., no infinite dependency chain contains infinitely many occurrences of the paths. Unfortunately, this idea cannot be implemented in the DP framework since the framework does not keep information of connections between two continuous dependency pairs (i.e., edges of dependency graphs).

In this paper, we propose methods for proving termination of constrained TRSs. We first extend both the termination criterion for TRSs w.r.t. dependency pairs and the DP framework to constrained TRSs, and then extend them so that dependency pair processors take a subgraph of the dependency graph as input and output a finite set of subgraphs obtained by eliminating nodes and/or edges from the input. For the sake of readability, we deal with the ordinary class of constrained TRSs [6,7,17,30,27]. It is straightforward to extend the results of this paper to more complicated classes such as CESs [13,14].

The first extension is straightforward, while we combine the lower bound detection [20] and the integer polynomial interpretation method [13,14] for CESs. In the second extension, we show how to adapt the existing sound and complete DP processors taking dependency pairs as input into the sound and complete processors for the extended DP framework, i.e., the extended framework is a strict extension of the ordinary one. Since paths from dependency pairs to their accident dependency pairs are captured as edges in the dependency graph, the extended framework can eliminate the paths mentioned above from the graph.

In [31], DP processors take dependency graphs as input in order to update graphs of which some nodes are not dependency pairs after the application of transformational processors. In this case, such graphs cannot be computed in keeping with the definition of dependency graphs and thus the processors related

to the update eliminate edges between dependency pairs and other kinds of pairs. Therefore, the purpose and use of carrying edges is different from ours.

The main contribution of this paper is to extend the DP framework for constrained TRSs to the one in which DP processors handle dependency graphs and eliminate nodes and/or edges from the graphs. This extended framework is applicable to the case of unconstrained TRSs.

This paper is organized as follows. Section 2 prepares notation of constrained TRSs. Section 3 extends the termination criterion for TRSs w.r.t. dependency pairs to constrained TRSs. Section 4 shows a necessary condition of two dependency pairs for forming a dependency chain. Section 5 extends the DP framework to constrained TRSs. Section 6 extends the DP framework to the one handling edges of dependency graphs. Section 7 concludes and shows future work of this research. Missing proofs of technical results can be found in the appendix of the full version of this paper [29].

## 2 Preliminaries

In this section, we recall some basic notions and notations of term rewriting [3, 28], the first-order predicate logic [24], and constrained TRSs [6, 7, 17, 30, 27].

Throughout this paper, we use  $\mathcal{V}$  as a countably infinite set of variables. The set of *terms* over a signature  $\mathcal{F}$  and  $\mathcal{V}$  is denoted by  $T(\mathcal{F}, \mathcal{V})$ . The set of variables appearing in a term  $t$  is denoted by  $\text{Var}(t)$ . The *identity* of terms  $s$  and  $t$  is denoted by  $s \equiv t$ . For a term  $t$  and a *position*  $p$  of  $t$ , the notation  $t|_p$  represents the *subterm* of  $t$  at  $p$ . We write  $s \supseteq t$  if  $s|_p \equiv t$ , and write  $s \triangleright t$  if  $s|_p \equiv t$  and  $p \neq \varepsilon$ . If position  $p$  is a proper prefix of a position  $q$ , then we write  $p < q$ . The symbol at the *root* position  $\varepsilon$  of  $t$  is denoted by  $\text{root}(t)$ . For a *context*  $C[\ ]$  with  $n$ -holes at positions  $p_1, \dots, p_n$ , the notation  $C[t_1, \dots, t_n]_{p_1, \dots, p_n}$  represents the term obtained by replacing hole  $\square$  at position  $p_i$  with term  $t_i$  for  $1 \leq i \leq n$ . The *domain* and *range* of a *substitution*  $\sigma$  are denoted by  $\text{Dom}(\sigma)$  and  $\text{Ran}(\sigma)$ , respectively. The application of  $\sigma$  to term  $t$  is abbreviated to  $t\sigma$ . If  $\text{Dom}(\sigma) = \{x_1, \dots, x_n\}$ , then we may write  $\{x_i \mapsto \sigma(x_i) \mid 1 \leq i \leq n\}$  instead of  $\sigma$ . The restriction  $\sigma|_X$  of  $\sigma$  to a set  $X \subseteq \mathcal{V}$  is defined as  $\sigma|_X = \{x \mapsto \sigma(x) \mid x \in \text{Dom}(\sigma) \cap X\}$ .

Let  $\mathcal{G}$  be a signature such that  $\mathcal{G}$  has at least a constant (i.e.,  $T(\mathcal{G}) \neq \emptyset$ ), and  $\mathcal{P}$  be a finite set of *predicate* symbols. *Formulas* over  $(\mathcal{G}, \mathcal{P}, \mathcal{V})$  have the following syntax given in Backus Naur form:  $\phi ::= P(t_1, \dots, t_n) \mid \top \mid \perp \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi)$  where  $P$  is an  $n$ -ary predicate in  $\mathcal{P}$  and  $t_1, \dots, t_n \in T(\mathcal{G}, \mathcal{V})$ . We sometimes denote  $(\neg\phi_1 \vee \phi_2)$  by  $(\phi_1 \Rightarrow \phi_2)$  as usual. We often abbreviate brackets in formulas as usual. The set of *free variables* in a formula  $\phi$  is denoted by  $\text{fv}(\phi)$ . A formula  $\phi$  is called *closed* if  $\text{fv}(\phi) = \emptyset$ . Substitutions from  $\mathcal{V}$  to  $T(\mathcal{G}, \mathcal{V})$  are applied to formulas as usual. Note that, given another signature  $\mathcal{F}$ , we allow to apply substitution  $\sigma$  from  $\mathcal{V}$  to  $T(\mathcal{F} \cup \mathcal{G}, \mathcal{V})$ , to formulas  $\phi$ , denoted by  $\phi\sigma$ , if  $\text{Ran}(\sigma|_{\text{fv}(\phi)}) \subseteq T(\mathcal{G}, \mathcal{V})$ . A *structure*  $\mathcal{M}$  for  $(\mathcal{G}, \mathcal{P}, \mathcal{V})$  is a triple  $(\mathcal{A}, \mathcal{G}^{\mathcal{M}}, \mathcal{P}^{\mathcal{M}})$  such that the *universe*  $\mathcal{A}$  is a non-empty set of concrete values,  $\mathcal{G}^{\mathcal{M}}$  is a set of mappings where  $g^{\mathcal{M}} : \mathcal{A}^n \rightarrow \mathcal{A}$  is included in  $\mathcal{G}^{\mathcal{M}}$  for every  $n$ -ary function symbol  $g \in \mathcal{G}$ , and  $\mathcal{P}^{\mathcal{M}}$  is a family of sets where  $P^{\mathcal{M}} \subseteq \mathcal{A}^n$  is included in  $\mathcal{P}^{\mathcal{M}}$  for

every  $n$ -ary predicate symbol  $P \in \mathcal{P}$ . For a closed formula  $\phi$ , we write  $\mathcal{M} \models \phi$  if  $\phi$  holds w.r.t.  $\mathcal{M}$  as usual. A formula  $\phi$  is called *valid* (*satisfiable*, resp.) *w.r.t.*  $\mathcal{M}$  if  $\mathcal{M} \models \phi\theta$  for every (some, resp.) substitution  $\theta$  such that  $\text{Ran}(\theta|_{\text{fv}(\phi)}) \subseteq T(\mathcal{G})$ . When  $\mathcal{G}$ ,  $\mathcal{P}$ , and  $\mathcal{M}$  are fixed in context, we may call formulas over  $(\mathcal{G}, \mathcal{P}, \mathcal{V})$  *constraints* (w.r.t.  $\mathcal{M}$ ). We suppose that the binary predicate symbol  $\simeq$ , so-called the *equality predicate*, is included in  $\mathcal{P}$  and its interpretation  $\simeq^{\mathcal{M}}$  is the identity over  $\mathcal{A}$ , i.e.,  $\simeq^{\mathcal{M}} = \{(a, a) \mid a \in \mathcal{A}\}$ .

*Example 1.* Let  $\mathcal{G}_{PA}$  be a signature  $\{0, \text{s}(), \text{p}(), \text{plus}(), \text{minus}(), \}$ ,  $\mathcal{P}_{PA}$  be a set  $\{\succ(\cdot, \cdot), \simeq(\cdot, \cdot)\}$  of predicate symbols, and  $\mathcal{M}_{PA}$  be a structure  $(\mathbb{Z}, \{0^{\mathcal{M}_{PA}}, \text{s}^{\mathcal{M}_{PA}}, \text{p}^{\mathcal{M}_{PA}}, \text{plus}^{\mathcal{M}_{PA}}, \text{minus}^{\mathcal{M}_{PA}}\}, \{\succ^{\mathcal{M}_{PA}}, \simeq^{\mathcal{M}_{PA}}\})$  for  $(\mathcal{G}_{PA}, \mathcal{P}_{PA}, \mathcal{V})$  where  $\mathbb{Z}$  is the set of integers,  $0^{\mathcal{M}_{PA}} = 0$ ,  $\text{s}^{\mathcal{M}_{PA}}(x) = x + 1$ ,  $\text{p}^{\mathcal{M}_{PA}}(x) = x - 1$ ,  $\text{plus}^{\mathcal{M}_{PA}}(x, y) = x + y$ ,  $\text{minus}^{\mathcal{M}_{PA}}(x, y) = x - y$ , and  $\succ^{\mathcal{M}_{PA}} = \{(n, m) \mid n > m\}$ .

Let  $\mathcal{M}$  be a structure for  $(\mathcal{G}, \mathcal{P}, \mathcal{V})$  and  $\mathcal{F}$  be a signature such that  $\mathcal{F} \cap \mathcal{G} = \emptyset$ . A *constrained rewrite rule* over  $(\mathcal{F}, \mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{M})$  is a triple  $(l, r, \phi)$ , written as  $l \rightarrow r \llbracket \phi \rrbracket$ , such that  $l$  and  $r$  are terms in  $T(\mathcal{F} \cup \mathcal{G}, \mathcal{V})$ ,  $l$  is not a variable,  $\phi$  is a constraint w.r.t.  $\mathcal{M}$  (i.e., a formula over  $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ ),  $\text{Var}(l) \supseteq \text{Var}(r) \cup \text{fv}(\phi)$ , and  $\phi$  is satisfiable w.r.t.  $\mathcal{M}$ .<sup>1</sup> We may write  $l \rightarrow r$  instead of  $l \rightarrow r \llbracket \top \rrbracket$ .

A *constrained term rewriting system*  $R$  (constrained TRS, for short) is a finite set of constrained rewrite rules over  $(\mathcal{F}, \mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{M})$ . Unless noted otherwise,  $R$  is a constrained TRS over  $(\mathcal{F}, \mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{M})$ . The rewrite relation  $\rightarrow_R$  of  $R$  is defined as follows:  $\rightarrow_R = \{(C[l\sigma]_p, C[r\sigma]_p) \mid l \rightarrow r \llbracket \phi \rrbracket \in R, \text{Ran}(\sigma|_{\text{fv}(\phi)}) \subseteq T(\mathcal{G}, \mathcal{V}), \phi\sigma \text{ is valid w.r.t. } \mathcal{M}\}$ . To specify position  $p$  above explicitly for  $C[s\sigma]_p \rightarrow_R C[t\sigma]_p$ , we may write  $\rightarrow_R^p$  instead of  $\rightarrow_R$ . Moreover, we may write  $\rightarrow_R^{\varepsilon < p}$  instead of  $\rightarrow_R$  or  $\rightarrow_R^p$  if  $\varepsilon < p$ . A term  $t$  is called *terminating* w.r.t.  $\rightarrow_R$  if there is no infinite rewrite sequence of  $\rightarrow_R$  that is starting from  $t$ . The constrained TRS  $R$  is called *terminating* if every term is terminating w.r.t.  $\rightarrow_R$ . We say that  $R$  is *locally sound w.r.t. its structure* (locally sound, for short) if for every pair of terms  $s \in T(\mathcal{G}, \mathcal{V})$  and  $t \in T(\mathcal{F} \cup \mathcal{G})$ ,  $s \rightarrow_R t$  implies both  $t \in T(\mathcal{G}, \mathcal{V})$  and  $s \simeq t$  is valid w.r.t.  $\mathcal{M}$  (i.e., for all  $l \rightarrow r \llbracket \phi \rrbracket$  in  $R$ ,  $l \in T(\mathcal{G}, \mathcal{V})$  implies both  $r \in T(\mathcal{G}, \mathcal{V})$  and  $\phi \Rightarrow l \simeq r$  is valid w.r.t.  $\mathcal{M}$ ). Roughly speaking, local soundness guarantees that the reduction over  $T(\mathcal{G})$  is consistent with the structure  $\mathcal{M}$ .

*Example 2.* Consider the C program and the locally sound constrained TRS  $R_1$  over  $(\{\text{f}(), \text{u}(), \cdot, \cdot, \cdot\}, \mathcal{G}_{PA}, \mathcal{P}_{PA}, \mathcal{V}, \mathcal{M}_{PA})$  illustrated in Fig. 1 where the TRS  $R_1$  is obtained from the C program [17]. Note that  $R_1$  is a variant of the constrained equational system obtained from the C program [14].

<sup>1</sup> Constrained rewrite rules employed in [6, 7] are assumed to be *linear* in order to use *tree automaton* and *context-free grammar* techniques [11]. However, this paper does not assume that constrained rewrite rules are *linear* and thus our results are useful in proving termination of linear constrained TRSs in [6, 7]. On the other hand, auxiliary constrained rules in CESs [13] for interpreted symbols are assumed to be *right-linear* because of the use of *equational rewriting*. Since this paper does not deal with equational rewriting, our constrained rules for interpreted symbols are not assumed to be right-linear.

$$\begin{array}{l}
\text{int } f(\text{int } x)\{ \\
\quad \text{int } i = 0, z = 1; \\
\quad \text{while}( x > i )\{ \\
\quad \quad z += f(i); \\
\quad \quad i++; \\
\quad \} \\
\quad \text{return } z; \\
\}
\end{array}
\quad R_1 = \left\{ \begin{array}{ll}
f(x) \rightarrow u(x, 0, s(0)) & \\
u(x, i, z) \rightarrow u(x, s(i), \text{plus}(z, f(i))) & \llbracket x \succ i \rrbracket \\
u(x, i, z) \rightarrow z & \llbracket \neg(x \succ i) \rrbracket \\
\text{plus}(x, y) \rightarrow x & \llbracket y \simeq 0 \rrbracket \\
\text{plus}(x, s(y)) \rightarrow s(\text{plus}(x, y)) & \llbracket s(y) \succ 0 \rrbracket \\
\text{plus}(x, p(y)) \rightarrow p(\text{plus}(x, y)) & \llbracket 0 \succ p(y) \rrbracket \\
\text{minus}(x, y) \rightarrow x & \llbracket y \simeq 0 \rrbracket \\
\text{minus}(x, s(y)) \rightarrow p(\text{minus}(x, y)) & \llbracket s(y) \succ 0 \rrbracket \\
\text{minus}(x, p(y)) \rightarrow s(\text{minus}(x, y)) & \llbracket 0 \succ p(y) \rrbracket \\
s(p(x)) \rightarrow x & p(s(x)) \rightarrow x
\end{array} \right\}$$

Fig. 1. A C program and the corresponding constrained TRS

### 3 Termination Criterion for Constrained TRSs

In this section, we extend the termination criterion for TRSs [2], which is based on *dependency pairs*, to constrained TRSs. Let  $R$  be a constrained TRS. The set of *defined symbols* of  $R$  is denoted by  $\mathcal{D}_R$ , i.e.,  $\mathcal{D}_R = \{\text{root}(l) \in \mathcal{F} \cup \mathcal{G} \mid l \rightarrow r \llbracket \phi \rrbracket \in R\}$ . The *marked* symbol of a defined symbol  $f \in \mathcal{D}_R$  is denoted by  $f^\sharp$ . The set of marked symbols is denoted by  $\mathcal{D}_R^\sharp$ . For a term  $f(t_1, \dots, t_n)$  with  $f \in \mathcal{D}_R$ ,  $f(t_1, \dots, t_n)^\sharp$  denotes the term  $f^\sharp(t_1, \dots, t_n)$ .

We extend dependency pairs and chains for TRSs to constrained TRSs.

**Definition 3.** Let  $R$  be a constrained TRS. A *dependency pair* of  $R$  is a constrained rewrite rule  $l^\sharp \rightarrow t^\sharp \llbracket \phi \rrbracket$  over  $(\mathcal{F} \cup \mathcal{D}_R^\sharp, \mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{M})$  where  $l \rightarrow r \llbracket \phi \rrbracket \in R$ ,  $t$  is a subterm of  $r$  such that  $\text{root}(t) \in \mathcal{D}_R$  and  $t$  is not a proper subterm of  $l$ . The set of dependency pairs of  $R$  is denoted by  $DP(R)$ .

*Example 4.* The dependency pairs of  $R_1$  in Fig. 1 are illustrated in Fig. 2.

**Definition 5.** Let  $R$  be a constrained TRS and  $S \subseteq DP(R)$ . A (possibly infinite) sequence  $s_1 \rightarrow t_1 \llbracket \phi_1 \rrbracket, s_2 \rightarrow t_2 \llbracket \phi_2 \rrbracket, \dots$  of *dependency pairs* in  $S$  is called an *S-chain* (of  $R$ ) if there are substitutions  $\sigma_1, \sigma_2, \dots$  such that  $t_i \sigma_i \xrightarrow{\varepsilon^<}_R s_{i+1} \sigma_{i+1}$ ,  $\text{Ran}(\sigma|_{\mathcal{F}(\phi_i)}) \subseteq T(\mathcal{G}, \mathcal{V})$ , and  $\phi_i \sigma_i$  is valid w.r.t.  $\mathcal{M}$  for all  $i \geq 1$ . Notice that if the length of the sequence is  $n$  then  $\phi_n \sigma_n$  is valid w.r.t.  $\mathcal{M}$ . The chain is called *minimal* if  $t_i \sigma_i$  is terminating w.r.t.  $\rightarrow_R$  for all  $i \geq 1$ . Moreover, the infinite chain is called *S-innumerable* (*S-innumerable chain*, for short) if every element in  $S$  appears infinitely many times in the chain.

A main difference from chains of TRSs is validity of  $\phi_i \sigma_i$ . Moreover, when a chain is finite, we impose validity of the constraint of the last dependency pair of the chain. This is because the chain  $f^\sharp(s_1, \dots, s_n) \rightarrow t \llbracket \phi \rrbracket, u \rightarrow g^\sharp(v_1, \dots, v_m) \llbracket \psi \rrbracket$  means that  $f$  calls  $g$ . For the case of TRSs,  $\phi$  and  $\psi$  are  $\top$  and thus their validity is always guaranteed.

The termination criterion for TRSs [2], which is based on dependency pairs, is extended to constrained TRSs as follows.

<sup>2</sup> For the sake of readability, we employ the original definition of chains [2], while it is easy to extend it to a more advanced one [19].

$$DP(R_1) = \left\{ \begin{array}{ll} (1) & f^\sharp(x) \rightarrow u^\sharp(x, 0, s(0)) \\ (2) & f^\sharp(x) \rightarrow s^\sharp(0) \\ (3) & u^\sharp(x, i, z) \rightarrow u^\sharp(x, s(i), \text{plus}(z, f(i))) \quad [x \succ i] \\ (4) & u^\sharp(x, i, z) \rightarrow s^\sharp(i) \quad [x \succ i] \\ (5) & u^\sharp(x, i, z) \rightarrow \text{plus}^\sharp(z, f(i)) \quad [x \succ i] \\ (6) & u^\sharp(x, i, z) \rightarrow f^\sharp(i) \quad [x \succ i] \\ (7) & \text{plus}^\sharp(x, s(y)) \rightarrow s^\sharp(\text{plus}(x, y)) \quad [s(y) \succ 0] \\ (8) & \text{plus}^\sharp(x, s(y)) \rightarrow \text{plus}^\sharp(x, y) \quad [s(y) \succ 0] \\ (9) & \text{plus}^\sharp(x, p(y)) \rightarrow p^\sharp(\text{plus}(x, y)) \quad [0 \succ p(y)] \\ (10) & \text{plus}^\sharp(x, p(y)) \rightarrow \text{plus}^\sharp(x, y) \quad [0 \succ p(y)] \\ (11) & \text{minus}^\sharp(x, s(y)) \rightarrow p^\sharp(\text{minus}(x, y)) \quad [s(y) \succ 0] \\ (12) & \text{minus}^\sharp(x, s(y)) \rightarrow \text{minus}^\sharp(x, y) \quad [s(y) \succ 0] \\ (13) & \text{minus}^\sharp(x, p(y)) \rightarrow s^\sharp(\text{minus}(x, y)) \quad [0 \succ p(y)] \\ (14) & \text{minus}^\sharp(x, p(y)) \rightarrow \text{minus}^\sharp(x, y) \quad [0 \succ p(y)] \end{array} \right\}$$

**Fig. 2.** The dependency pairs of  $R_1$

**Theorem 6.** *A constrained TRS  $R$  is terminating iff there exists no infinite minimal  $DP(R)$ -chain.*

*Proof.* The proof is similar to the case of TRSs (cf. Theorem 6 in [2]).  $\square$

To prove termination of a constrained TRS  $R$ , we may show that there is no infinite minimal  $DP(R)$ -chain. To this end, we show a criterion for an  $S \subseteq DP(R)$  to reduce the non-existence of infinite minimal  $S$ -chains to the non-existence of both  $S$ -innumerable chains and infinite minimal  $S'$ -chains for some of proper subsets  $S'$  of  $S$ . Let  $A$  be a set. The *power set* of  $A$  is denoted by  $\mathcal{Power}(A)$ . A set  $B$  in  $\mathcal{Power}(A)$  is called *maximal in*  $X \subseteq \mathcal{Power}(A)$  if  $B = C$  whenever  $B \subseteq C$  for any  $C \in X$ .

**Theorem 7.** *Let  $R$  be a constrained TRS,  $S \subseteq DP(R)$  and  $X \subseteq \mathcal{Power}(S)$  such that there exists no  $S'$ -innumerable chain for any  $S' \in X$ . There exists no infinite (minimal)  $S$ -chain iff there exists no infinite (minimal)  $S'$ -chain for any maximal  $S'$  in  $\mathcal{Power}(S) \setminus X$ .*

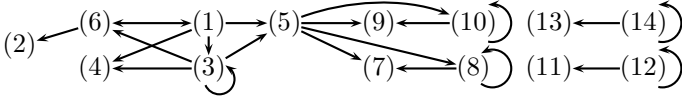
The proof can be found in [29]. Thanks to Theorem 7, to prove the non-existence of infinite chains, we may show that for some  $X \subseteq \mathcal{Power}(S)$ ,

- there is no  $S'$ -innumerable chain for any  $S'$  in  $X$ , and
- there is no infinite minimal  $S'$ -chain for any maximal set  $S'$  in  $\mathcal{Power}(S) \setminus X$ .

This scheme is helpful to show soundness of DP processors shown later.

Finally, we explain how Theorem 7 helps us to show termination, by extending Theorem 18 in [2] to constrained TRSs.

**Definition 8.** *Let  $R$  be a constrained TRS and  $S \subseteq DP(R)$ . The dependency graph of  $S$  and  $R$ , written as  $DG(S, R)$ , is a directed graph  $(V, E)$  such that  $V = S$  and  $E = \{(u, v) \mid u, v \in V, \text{ the sequence “}u, v\text{” is an } S\text{-chain}\}$ . The dependency graph of  $DP(R)$  and  $R$  is called the dependency graph of  $R$ , and we denote  $DG(DP(R), R)$  by  $DG(R)$ .*



**Fig. 3.** The dependency graph of  $R_1$

**Theorem 9.** *Let  $R$  be a constrained TRS and  $S \subseteq DP(R)$ . There exists no infinite (minimal)  $S$ -chain iff there exists no infinite (minimal)  $S'$ -chain for any  $S'$  that is the vertex set of a strongly connected component (SCC, for short) of  $DG(S, R)$ .*

*Proof.* Let  $X = \{S' \mid DG(S', R) \text{ is not an SCC of } DG(S, R)\}$ . Then, this theorem follows from Theorem 7 since all of the following hold clearly:

- there exists no  $S'$ -innumerable chain for any  $S' \in X$ , and
- maximal sets in a  $\mathcal{Power}(S) \setminus X$  are vertex sets of SCCs of  $DG(S, R)$ .  $\square$

*Example 10.* The dependency graph of  $R_1$  in Fig. 2 is illustrated in Fig. 3. The vertex sets of SCCs of  $DG(R_1)$  are  $\{(1), (3), (6)\}$ ,  $\{(8)\}$ ,  $\{(10)\}$ ,  $\{(12)\}$  and  $\{(14)\}$ . To prove termination of  $R_1$ , it suffices to show the non-existence of infinite minimal chains for all of them.

As for the case of TRSs, the dependency graph is not computable for every constrained TRS. Thus, for constrained TRSs, we employ approximation techniques of dependency graphs for TRSs since for any constrained TRS  $R$ , the dependency graph of the TRS obtained from  $R$  by removing constraints from rewrite rules is an over approximation of  $DG(R)$ . In the next section, we will show one of the approaches to improve over approximated dependency graphs.

## 4 Necessary Condition of Two Dependency Pairs for Forming a Chain

In this section, for locally sound constrained TRSs, we present a necessary condition of two dependency pairs  $s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket$  for forming a chain. For two terms  $t, u$  (the right-hand side of  $s \rightarrow t \llbracket \phi \rrbracket$  and the left-hand side of  $u \rightarrow v \llbracket \psi \rrbracket$ ) and for a set  $V$  of variables, we construct a constraint  $\eta$  such that for any substitutions  $\sigma, \theta$  with  $\mathcal{Ran}(\sigma|_{V \cup \mathcal{FV}(\phi)}) \subseteq T(\mathcal{G}, \mathcal{V})$ , the constraint  $\eta(\sigma \cup \theta)$  is valid if  $t\sigma \xrightarrow{*} u\theta$ . Thus, satisfiability of  $\phi \wedge \eta \wedge \psi$  is a necessary condition of the two dependency pairs for forming a chain. The set  $V$  is used to specify variables that are substituted by terms in  $T(\mathcal{G}, \mathcal{V})$ .

First, we show how to construct the constraint  $\eta$  mentioned above.

**Definition 11.** *Let  $R$  be a locally sound constrained TRS,  $V \subseteq \mathcal{V}$ ,  $t$  and  $u$  be terms in  $T(\mathcal{F} \cup \mathcal{G} \cup \mathcal{D}_R^\sharp, \mathcal{V})$  such that  $t \equiv C[t_1, \dots, t_n]$ ,  $u \equiv C[u_1, \dots, u_n]$ , and  $\text{root}(t_i) = \text{root}(u_i)$  implies  $\text{root}(t_i) \in \mathcal{D}_R$  for some context  $C[\ ]$  over  $T(\mathcal{C}_R \cup \mathcal{D}_R^\sharp \cup$*



$\{\square\}, \mathcal{V})$ . Note that for any pair of terms  $t$  and  $u$ , the context  $C[\ ]$  is unique. The constraint  $\text{reach}_{R,V}(C[t_1, \dots, t_n], C[u_1, \dots, u_n])$  is defined as  $\bigwedge_{i=1}^n b_i$  such that

$$b_i = \begin{cases} t_i \simeq u_i & \text{if } t_i \in T(\mathcal{G}, V), u_i \in T(\mathcal{G}, \mathcal{V}), \text{ and } \text{root}(t_i) \in \mathcal{C}_R \text{ implies } u_i \in \mathcal{V} \\ \perp & \text{if } t_i \in T(\mathcal{G}, V) \text{ and } u_i \notin T(\mathcal{G}, \mathcal{V}) \\ \perp & \text{if } \text{root}(t_i) \in \mathcal{C}_R \cup \mathcal{D}_R^\# \text{ and } u_i \notin \mathcal{V} \\ \top & \text{otherwise} \end{cases}$$

For the first case of the construction of  $b_i$ , we let  $b_i$  be the formula  $t_i \simeq u_i$  since  $R$  is locally sound and then for all substitutions  $\sigma$  and  $\theta$ , the constraint  $t_i\sigma \simeq u_i\theta$  is valid w.r.t.  $\mathcal{M}$  whenever  $t_i\sigma \xrightarrow{*}_R u_i\theta$  and  $\mathcal{Ran}(\sigma|_V) \subseteq T(\mathcal{G}, \mathcal{V})$ . For the second and third cases, we let  $b_i$  be the invalid constraint  $\perp$  since any instance of  $t_i$  cannot be reduced to some instance of  $u_i$ . For any other cases, we let  $b_i$  be  $\top$  since there may exist substitutions  $\sigma$  and  $\theta$  such that  $t_i\sigma \xrightarrow{*}_R u_i\theta$ . Since  $C[\ ]$  is a constructor context of  $R$  and then the establishment of  $C[t_1, \dots, t_n]\sigma \xrightarrow{*}_R C[u_1, \dots, u_n]\theta$  coincides with that of all the subderivations  $t_1\sigma \xrightarrow{*}_R u_1\theta, \dots, t_n\sigma \xrightarrow{*}_R u_n\theta$ , we construct the constraint  $\bigwedge_{i=1}^n b_i$  for terms  $t$  and  $u$ . In summary,  $t\sigma \xrightarrow{*}_R u\theta$  implies validity of the constraint  $(\text{reach}_{R,V}(t, u))(\sigma \cup \theta)$ , i.e., unsatisfiability of  $\text{reach}_{R,V}(t, u)$  implies non-existence of substitutions  $\sigma$  and  $\theta$  such that  $t\sigma \xrightarrow{*}_R u\theta$ .

**Lemma 12.** *Let  $R$  be a locally sound constrained TRS,  $t$  and  $u$  be terms over  $T(\mathcal{F} \cup \mathcal{G} \cup \mathcal{D}_R^\#, \mathcal{V})$  such that  $\text{Var}(t) \cap \text{Var}(u) = \emptyset$ ,  $V$  be a subset of  $\mathcal{V}$  such that  $\text{Var}(u) \cap V = \emptyset$ , and  $\sigma, \theta$  be substitutions such that  $\mathcal{Ran}(\sigma|_V) \subseteq T(\mathcal{G}, \mathcal{V})$ . If  $\sigma \xrightarrow{*}_R t\theta$ , then*

- $\mathcal{Ran}((\sigma \cup \theta)|_{\text{fv}(\text{reach}_{R,V}(t, u))}) \subseteq T(\mathcal{G}, \mathcal{V})$ , and
- $(\text{reach}_{R,V}(s, t))(\sigma \cup \theta)$  is valid w.r.t.  $\mathcal{M}$ .

The proof can be found in [29].

*Example 13.* Consider the constrained TRS  $R_1$  in Fig. 2 and terms  $\text{plus}^\#(x, y)$  and  $\text{plus}^\#(x', \text{p}(y'))$ . We have  $\text{reach}_{R_1, \{y\}}(\text{plus}^\#(x, y), \text{plus}^\#(x', \text{p}(y'))) = \top \wedge y \simeq \text{p}(y')$ . Let  $\sigma$  and  $\theta$  be substitutions such that  $y\sigma \in T(\mathcal{G}_{PA}, \mathcal{V})$ . Then,  $\top(\sigma \cup \theta)$  is valid w.r.t.  $\mathcal{M}_{PA}$  if  $x\sigma \xrightarrow{*}_{R_1} x'\theta$ . Moreover,  $(y \simeq \text{p}(y'))(\sigma \cup \theta)$  is also valid w.r.t.  $\mathcal{M}_{PA}$  if  $y\sigma \xrightarrow{*}_{R_1} \text{p}(y')\theta$  since  $R_1$  is locally sound. Thus, the constraint  $(\text{reach}_{R_1, \{y\}}(\text{plus}^\#(x, y), \text{plus}^\#(x', \text{p}(y'))))(\sigma \cup \theta)$  is valid w.r.t.  $\mathcal{M}_{PA}$  if  $\text{plus}^\#(x, y)\sigma \xrightarrow{*}_{R_1} \text{plus}^\#(x', \text{p}(y'))\theta$ . Consider terms  $\text{plus}^\#(x, y)$  and  $\text{f}^\#(x')$ . We have  $\text{reach}_{R_1, \{y\}}(\text{plus}^\#(x, y), \text{f}^\#(x')) = \perp$ , an unsatisfiable constraint. The unsatisfiability tells us that there is no substitutions  $\sigma$  and  $\theta$  such that  $\text{plus}^\#(x, y)\sigma \xrightarrow{*}_{R_1} \text{f}^\#(x')\theta$ .

Next we define a constraint of which satisfiability is a necessary condition of two dependency pairs for forming a chain.

**Definition 14.** *Let  $R$  be a constrained TRS,  $s \rightarrow t \llbracket \phi \rrbracket$ ,  $u \rightarrow v \llbracket \psi \rrbracket \in DP(R)$ , and  $V \subseteq \text{Var}(t)$ . Then, we denote the constraint  $\phi \wedge \text{reach}_{R, V \cup \text{fv}(\phi)}(t, u) \wedge \psi$  by  $\text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket, V)$ .*



**Theorem 15.** *Let  $R$  be a locally sound constrained TRS,  $s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket \in DP(R)$ ,  $u' \rightarrow v' \llbracket \psi' \rrbracket$  be a renamed variant of  $u \rightarrow v \llbracket \psi \rrbracket$  such that  $\text{Var}(s) \cap \text{Var}(u') = \emptyset$ ,  $V$  be a subset of  $\text{Var}(t)$ , and  $\sigma, \theta$  be substitutions such that  $\text{Ran}(\sigma|_{V \cup \text{fv}(\phi) \cup \text{fv}(\psi')}) \subseteq T(\mathcal{G}, \mathcal{V})$ . If  $\phi\sigma$  and  $\psi'\theta$  are valid w.r.t.  $\mathcal{M}$  and  $t\sigma \xrightarrow{*}_R v'\theta$  then*

- $\text{Ran}(\sigma|_{\text{fv}(\text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, V))}) \subseteq T(\mathcal{G}, \mathcal{V})$ , and
- $(\text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, V))(\sigma \cup \theta)$  is valid w.r.t.  $\mathcal{M}$ ,

*i.e., if the sequence  $s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket$  is a  $DP(R)$ -chain, then  $\text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \emptyset)$  is satisfiable w.r.t.  $\mathcal{M}$ .*

*Proof.* This theorem follows from Lemma 12. □

Thanks to Theorem 15, we obtain a sufficient condition of two dependency pairs that form no chain.

**Corollary 16.** *Let  $R$  be a locally sound constrained TRS,  $S \subseteq DP(R)$ ,  $s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket \in S$ ,  $u' \rightarrow v' \llbracket \psi' \rrbracket$  be a renamed variant of  $u \rightarrow v \llbracket \psi \rrbracket$  such that  $\text{Var}(s) \cap \text{Var}(u') = \emptyset$ . If  $\text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \emptyset)$  is unsatisfiable w.r.t.  $\mathcal{M}$ , then  $s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket$  is not an  $S$ -chain.*

*Example 17.* Consider the dependency pairs (8) and (10) of  $DP(R_1)$  in Fig. 2. Let (10)' be a renamed variant of (10) such that  $x$  and  $y$  are renamed into  $x'$  and  $y'$ , respectively. Then, we have  $\text{chain}((8), (10)', \emptyset) = \mathbf{s}(y) \succ 0 \wedge (\top \wedge y \simeq \mathbf{p}(y')) \wedge 0 \succ \mathbf{p}(y')$ . This constraint corresponds to  $y + 1 > 0 \wedge y = y' - 1 \wedge 0 > y' - 1$  over integers by means of  $\mathcal{M}_{PA}$ , and this is unsatisfiable. Thus, we can know that (8), (10) is not a  $DP(R_1)$ -chain and then there is no edge from (8) to (10).

The sufficient condition in Corollary 16 for two dependency pairs that form no chain when constrained TRSs are locally sound is useful for improving over-approximated dependency graphs in employing approximation techniques for TRSs, i.e., if  $\text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \emptyset)$  is unsatisfiable w.r.t.  $\mathcal{M}$  then we may remove an edge  $(s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket)$  from over-approximated graphs. The other possible approach to the improvement of over-approximated dependency graphs is the one via the extension of unification in computing edges [13, 14]. However, we did not take this approach since constraints corresponding to the necessary condition are useful in developing DP processors shown later.

## 5 DP Framework for Constrained TRSs

In this section, we extend the *DP framework* for TRSs [19] to constrained TRSs.

Let  $R$  be a constrained TRS. A subset of  $DP(R)$  is called a *dependency pair problem* (of  $R$ ) (DP problem, for short)<sup>3</sup>. A DP problem  $S$  is called *finite* if there is no infinite minimal  $S$ -chain, and called *infinite* if it is not finite or  $R$

<sup>3</sup> This paper does not use pairs of  $(S, R)$  as DP problems since the second component  $R$  is not modified anywhere.

is not terminating. A *dependency pair processor* (DP processor, for short) is a function *Proc* which takes a DP problem as input and returns a finite set of DP problems. A DP processor *Proc* is called *sound* if for any DP problem *S*, the problem *S* is finite whenever all DP problems in *Proc*(*S*) are finite. *Proc* is called *complete* if for any DP problem *S*, the problem *S* is infinite whenever *Proc*(*S*) contains infinite DP problems. Therefore, a termination proof starts with the initial DP problem *DP*(*R*) and applies sound DP processors until an empty set of DP problems is obtained.

**Theorem 18.** *A constrained TRS  $R$  is terminating iff the DP problem  $DP(R)$  is finite. Moreover,  $R$  is not terminating iff the DP problem  $DP(R)$  is infinite.*

*Proof.* This theorem follows from Theorem 6.  $\square$

As a useful tool to prove soundness and completeness of DP processors, we adapt Theorem 7 to DP processors.

**Theorem 19.** *Let  $R$  be a constrained TRS. A DP processor *Proc* is sound and complete if for any  $S \subseteq DP(R)$ , there exists no  $S'$ -innumerable chain for any  $S' \subseteq S$  such that  $S' \setminus S'' \neq \emptyset$  for all  $S'' \in Proc(S)$ .*

*Proof.* We only prove soundness of *Proc* since completeness is trivial. Assume that all problems in *Proc*(*S*) are finite. Let  $X = \{S' \mid \forall S'' \in Proc(S). S' \setminus S'' \neq \emptyset\}$  and there is no  $S'$ -innumerable chain for any  $S' \in X$ . Then,  $Power(S) \setminus X = \{S' \mid \exists S'' \in Proc(S). S' \subseteq S''\}$ . Thus, all maximal sets in  $Power(S) \setminus X$  are included in *Proc*(*S*). Therefore, it follows from Theorem 7 that *S* is finite.  $\square$

We extend the DP processor w.r.t. dependency graphs [2] to constrained TRSs.

**Theorem 20.** *Let  $R$  be a constrained TRS and  $S \subseteq DP(R)$ . Then, the DP processor *Proc*, which takes a DP problem *S* and outputs vertex sets of SCCs of  $DG(S, R)$ , is sound and complete.*

*Proof.* Let  $X = \{S' \mid \forall S'' \in Proc(S). S' \setminus S'' \neq \emptyset\}$ . There exists no  $S'$ -innumerable chain for any  $S' \in X$  since  $S'$  is not a vertex set of strongly connected graphs which are included in  $DG(S, R)$ . Thus, it follows from Theorem 19 that the processor is sound and complete.  $\square$

Note that we can employ over approximation techniques for computing dependency graphs in Theorem 20.

The DP processor based on the *subterm criterion* [22] can be straightforwardly extended to constrained TRSs, by ignoring the constraints of dependency pairs.

**Theorem 21.** *Let  $R$  be a constrained TRS,  $S \subseteq DP(R)$ ,  $\pi$  be a simple projection from  $\mathcal{D}_R^\#$  to natural numbers such that  $1 \leq \pi(f^\#) \leq n$  for any  $n$ -ary marked symbol  $f^\# \in \mathcal{D}_R^\#$ . Moreover, let  $\triangleright_\pi = \{s \rightarrow t \llbracket \phi \rrbracket \mid s|_{\pi(\text{root}(s))} \triangleright t|_{\pi(\text{root}(t))}\}$  and  $\triangleright_\pi = \{s \rightarrow t \llbracket \phi \rrbracket \mid s|_{\pi(\text{root}(s))} \triangleright t|_{\pi(\text{root}(t))}\}$ . Then, the following processor *Proc* is sound and complete:*

$$Proc(S) = \begin{cases} \{S \setminus \triangleright_\pi\} & \text{if } S \subseteq \triangleright_\pi \\ \{S\} & \text{otherwise} \end{cases}$$

The proof can be found in [29].

*Example 22.* Consider  $DP(R_1)$  in Fig. 2. For the initial DP problem  $DP(R_1)$ , the DP processor in Theorem 20 converts  $DP(R_1)$  into  $\{\{(1), (3), (6)\}, \{(8)\}, \{(10)\}, \{(12)\}, \{(14)\}\}$ . Then, the DP processor in Theorem 21 transforms  $\{(8)\}, \{(10)\}, \{(12)\}$  and  $\{(14)\}$  into  $\{\emptyset\}$ . Thus, to prove termination of  $R_1$ , it suffices to show that  $\{(1), (3), (6)\}$  is finite.

In developing DP processors, it is useful to refer the constraints of dependency pairs. For example, the DP processor based on dependency graphs employs the constraints via the dependency graphs that are improved by the sufficient condition in Corollary 16, while the DP processor based the subterm criterion cannot employ the constraints. To show finiteness of the DP problem  $\{(1), (3), (6)\}$  in Example 22, the DP processor based on *polynomial interpretations over integers* [20, 22, 15] seems useful. Thus, in the rest of this section, we extend the DP processor based on polynomial interpretations to constrained TRSs.

The basic idea of the DP processor is to reduce an infinite ground derivation sequence obtained from an  $S$ -innumerable chain into an infinite strictly-decreasing sequence of integers with a lower bound. The lower bound leads us to a contradiction and hence the non-existence of  $S$ -innumerable chains. In ordering a dependency pair  $s \rightarrow t \llbracket \phi \rrbracket$  by  $>$  over polynomial interpretations and in detecting a lower bound for some  $s' \rightarrow t' \llbracket \phi' \rrbracket$ ,  $\phi$  and  $\phi'$  are sometimes useful, e.g., given a polynomial interpretation  $Pol$ , the validity of “ $\phi$  implies  $Pol(s) > Pol(t)$ ” and “ $\exists n \in \mathbb{Z}. \phi' \text{ implies } Pol(s') > n$ ” is enough. If the constraints of dependency pairs are over integer arithmetics, we can consider these conditions as arithmetic constraints over integers. To this end, we employ *tree homomorphisms* (cf. Chapter 1 in [11]) from marked symbols to the set of terms interpreted to integers.

**Definition 23.** Let  $\mathcal{F}_1$  and  $\mathcal{F}_2$  be sets of function symbols such that  $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$ . A tree homomorphism  $\mathcal{H}$  is a function from  $\mathcal{F}_1$  to  $T(\mathcal{F}_2, \mathcal{V})$  such that  $\mathcal{H}(f) \in T(\mathcal{F}_2, \{x_1, \dots, x_n\})$  for any  $n$ -ary function symbol  $f \in \mathcal{F}_1$ . The application of  $\mathcal{H}$  to terms in  $T(\mathcal{F}_1 \cup \mathcal{F}_2, \mathcal{V})$  is defined as follows:  $\mathcal{H}(x) = x$  for  $x \in \mathcal{V}$ ,  $\mathcal{H}(f(t_1, \dots, t_n)) = \mathcal{H}(f)\{x_i \mapsto \mathcal{H}(t_i) \mid 1 \leq i \leq n\}$  for  $f \in \mathcal{F}_1$ , and  $\mathcal{H}(g(t_1, \dots, t_n)) = g(\mathcal{H}(t_1), \dots, \mathcal{H}(t_n))$  for  $g \in \mathcal{F}_2$ .

Note that for a tree homomorphism  $\mathcal{H}$  from  $\mathcal{D}_R^\sharp$  to  $T(\mathcal{F} \cup \mathcal{G}, \mathcal{V})$  and terms  $t \in T(\mathcal{F} \cup \mathcal{G}, \mathcal{V})$  and  $f(t_1, \dots, t_n) \in T(\mathcal{F} \cup \mathcal{G}, \mathcal{V})$ , we have that  $\mathcal{H}(t) \equiv t$  and  $\mathcal{H}(f(t_1, \dots, t_n)^\sharp) \equiv \mathcal{H}(f^\sharp)\{x_i \mapsto t_i \mid 1 \leq i \leq n\}$ .

*Example 24.* Consider the term  $u^\sharp(x, s(i), \text{plus}(z, f(i)))$  in Fig. 2 and a tree homomorphism  $\mathcal{H} : \mathcal{D}_R^\sharp \rightarrow T(\mathcal{F} \cup \mathcal{G}_{PA}, \mathcal{V})$  such that  $\mathcal{H}(u^\sharp) = \text{minus}(x_1, x_2)$ . Then, we have  $\mathcal{H}(u^\sharp(x, s(i), \text{plus}(z, f(i)))) = \text{minus}(x, s(i))$ .

By imposing tree homomorphisms  $\mathcal{H}$  on a condition “ $\mathcal{H}(s), \mathcal{H}(t) \in T(\mathcal{G}, \mathcal{V})$  for all dependency pairs  $s \rightarrow t \llbracket \phi \rrbracket$ ” shown later, we can use negative integer as coefficients in polynomials, as well as [22, 20, 13, 16, 14].

Finally, we propose a simplified extension of the DP processor based on polynomial interpretations over integers. Let  $\mathcal{M}_{\mathbb{Z}}$  be a structure for  $(\mathcal{G}_{\mathbb{Z}}, \mathcal{P}_{\mathbb{Z}}, \mathcal{V})$  such that  $\succ \in P_{\mathbb{Z}}$ , the universe  $\mathbb{Z}$  is the set of integers and  $\succ^{\mathcal{M}_{\mathbb{Z}}} = \{n > m \mid n, m \in \mathbb{Z}\}$ .

**Theorem 25.** *Let  $R$  be a locally sound constrained TRS over  $(\mathcal{F}, \mathcal{G}_{\mathbb{Z}}, \mathcal{P}_{\mathbb{Z}}, \mathcal{V}, \mathcal{M}_{\mathbb{Z}})$ ,  $S \subseteq DP(R)$ , and  $\mathcal{H}$  be a tree homomorphism<sup>4</sup> from  $\mathcal{D}_R^{\sharp}$  to  $T(\mathcal{F} \cup \mathcal{G}_{\mathbb{Z}}, \mathcal{V})$  such that  $\mathcal{H}(s), \mathcal{H}(t) \in T(\mathcal{G}_{\mathbb{Z}}, \mathcal{V})$  for any  $s \rightarrow t \llbracket \phi \rrbracket \in S$ . Moreover, let*

- $S_{\succ} = \{s \rightarrow t \llbracket \phi \rrbracket \mid \forall u \rightarrow v \llbracket \psi \rrbracket \in S. \text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \text{Var}(\mathcal{H}(s))) \Rightarrow \mathcal{H}(s) \succ \mathcal{H}(t) \text{ is valid w.r.t. } \mathcal{M}_{\mathbb{Z}}\}$ ,
- $S_{\succsim} = \{s \rightarrow t \llbracket \phi \rrbracket \mid \forall u \rightarrow v \llbracket \psi \rrbracket \in S. \text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \text{Var}(\mathcal{H}(s))) \Rightarrow \mathcal{H}(s) \succ \mathcal{H}(t) \vee \mathcal{H}(s) \simeq \mathcal{H}(t) \text{ is valid w.r.t. } \mathcal{M}_{\mathbb{Z}}\}$ ,
- $S_{\text{bound}} = \{s \rightarrow t \llbracket \phi \rrbracket \mid \exists s_{\text{bound}} \in T(\mathcal{G}_{\mathbb{Z}}). \forall u \rightarrow v \llbracket \psi \rrbracket \in S. \text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \text{Var}(\mathcal{H}(s))) \Rightarrow \mathcal{H}(s) \succ s_{\text{bound}} \text{ is valid w.r.t. } \mathcal{M}_{\mathbb{Z}}\}$ ,
- $S_{\text{filter}} = \{s \rightarrow t \llbracket \phi \rrbracket \mid \text{Var}(\mathcal{H}(s)) \subseteq \text{fv}(\phi)\}$ , and
- $S_{\text{prsv}} = \{s \rightarrow t \llbracket \phi \rrbracket \mid \text{Var}(\mathcal{H}(t)) \subseteq \text{fv}(\phi) \cup \text{Var}(\mathcal{H}(s))\}$ ,

where  $u' \rightarrow v' \llbracket \psi' \rrbracket$  is a renamed variant of  $u \rightarrow v \llbracket \psi \rrbracket$  such that  $\text{Var}(s) \cap \text{Var}(u') = \emptyset$ . If none of  $S \cap S_{\succ}$ ,  $S \cap S_{\text{bound}}$  and  $S \cap S_{\text{filter}}$  is empty and  $S \subseteq S_{\succsim} \cap S_{\text{prsv}}$ , then there exists no  $S$ -innumerable chain, i.e., the following processor *Proc* is sound and complete:

$$\text{Proc}(S) = \begin{cases} \{S \setminus S_{\succ}, S \setminus S_{\text{bound}}, S \setminus S_{\text{filter}}\} & \text{if } S \subseteq S_{\succsim} \cap S_{\text{prsv}} \\ \{S\} & \text{otherwise} \end{cases}$$

The proof can be found in [29]. The sets  $S_{\text{filter}}$  and  $S_{\text{prsv}}$  in Theorem 25, original ones of this paper, guarantee that every  $S$ -innumerable chain with ground substitutions has a postfix chain from which we can obtain an infinite sequence of terms in  $T(\mathcal{G}_{\mathbb{Z}})$  by applying a tree homomorphism  $\mathcal{H}$  implementing a polynomial interpretation over integers. More precisely, for an  $S$ -innumerable chain  $s_1 \rightarrow t_1 \llbracket \phi_1 \rrbracket, s_2 \rightarrow t_2 \llbracket \phi_2 \rrbracket, \dots$  such that  $t_i \sigma_i \xrightarrow{*}_R s_{i+1} \sigma_{i+1}$ ,  $\phi_i \sigma_i$  is valid w.r.t.  $\mathcal{M}_{\mathbb{Z}}$  and  $s_i \sigma_i, t_i \sigma_i$  are ground,  $S_{\text{filter}}$  guarantees  $\mathcal{H}(s_k \sigma_k) \in T(\mathcal{G}_{\mathbb{Z}})$  for some  $k$ , and  $S_{\text{prsv}}$  implies  $\mathcal{H}(t_i \sigma_i), \mathcal{H}(s_{i+1} \sigma_{i+1}) \in T(\mathcal{G}_{\mathbb{Z}})$  for all  $j \geq k$ , i.e., we have an infinite sequence  $\mathcal{H}(s_k \sigma_k), \mathcal{H}(t_k \sigma_k), \mathcal{H}(s_{k+1} \sigma_{k+1}), \mathcal{H}(t_{k+1} \sigma_{k+1}), \dots$  of terms in  $T(\mathcal{G}_{\mathbb{Z}})$ . Moreover, local soundness of  $R$  implies that  $(\mathcal{H}(t_i \sigma_i))^{\mathcal{M}_{\mathbb{Z}}} = (\mathcal{H}(s_{i+1} \sigma_{i+1}))^{\mathcal{M}_{\mathbb{Z}}}$  for all  $i \geq k$ . For this reason, the DP processor needs not consider rewrite rules in  $R$ . The sets  $S_{\succsim}, S_{\succ}$  and  $S_{\text{bound}}$  in Theorem 25, variants of the usual ones, implies that  $(\mathcal{H}(s_i \sigma_i))^{\mathcal{M}_{\mathbb{Z}}} \geq (\mathcal{H}(t_i \sigma_i))^{\mathcal{M}_{\mathbb{Z}}}$  for all  $i \geq k$ , the strictly decreasing relation  $>$  appears infinitely many times, and there is a lower bound  $m (= (s_{\text{bound}})^{\mathcal{M}_{\mathbb{Z}}})$  such that  $(\mathcal{H}(s_i \sigma_i))^{\mathcal{M}_{\mathbb{Z}}} > m$  for all  $i \geq k$ . Thus, the  $S$ -innumerable chain leads us to a contradiction.

In Theorem 25, we restrict structures to ones with the set of integers as the universe. However, it is possible to generalize Theorem 25 to structures such that a binary predicate symbol  $\succ$  is interpreted to a *non-infinitesimal* one [20].

<sup>4</sup> In practical, we use tree homomorphisms  $\mathcal{H}$  such that  $\mathcal{H}(f^{\sharp})$  can be interpreted into a polynomial expression w.r.t. the given structure  $\mathcal{M}_{\mathbb{Z}}$  for any  $f \in \mathcal{D}_R^{\sharp}$ , e.g.,  $\mathcal{H}(u^{\sharp}) = \text{minus}(x_1, x_2)$  in Example 24.

*Example 26.* Consider the DP problem  $\{(1), (3), (6)\}$  in Example 22. Let  $\mathcal{H}$  be a tree homomorphism such that  $\mathcal{H}(f^\sharp) = \mathcal{H}(u^\sharp) = x_1$  and  $Proc$  be the DP processor in Theorem 25. Then,  $Proc(\{(1), (3), (6)\})$  is  $\{\{(3), (6)\}, \{(1)\}\}$  since both  $S_\succ$  and  $S_{\text{bound}}$  include (1), and  $S_{\text{filter}}$  includes both (3) and (6). We succeed in detecting a lower bound via (1). This is not possible if the processor does not refer the constraints of the dependency pairs following after (1), i.e., the case with  $S_{\text{bound}} = \{s \rightarrow t \llbracket \phi \rrbracket \mid \exists s_{\text{bound}} \in T(\mathcal{G}_{\mathbb{Z}}). \phi \Rightarrow \mathcal{H}(s) \succ s_{\text{bound}} \text{ is valid w.r.t. } \mathcal{M}_{\mathbb{Z}}\}$ . For this reason, a straightforward extension of the DP processor in [13, 14] to our constrained TRSs is not useful to show that the DP problem  $\{(1), (3), (6)\}$  is finite. After applying the DP processor in Theorem 20 to  $\{(3), (6)\}$  and  $\{(1)\}$ , the remaining DP problem is  $\{(3)\}$ . Given a tree homomorphism  $\mathcal{H}'$  such that  $\mathcal{H}'(u^\sharp) = \text{minus}(x_1, x_2)$ , the processor in Theorem 25 converts  $\{(3)\}$  to  $\{\emptyset\}$  and thus  $\{(1), (3), (6)\}$  is finite. Therefore,  $R_1$  is terminating. Note that termination of the C program in Fig 1 and of the corresponding integer rewriting system of  $R_1$  could not be proved by AProVE 1.2 [18, 16].

## 6 Graph-Handling DP Framework for Constrained TRSs

In this section, we extend the DP framework to the one such that DP processors handle subgraphs of dependency graphs, and show how to adapt the existing DP processors to such ones. Moreover, we extend the DP processor in Theorem 25 so that it eliminates none of nodes but some of edges.

*Example 27.* Consider the constrained TRS  $R_2$  such that  $R_2 = R_1 \cup \{u(x, i, z) \rightarrow u(p(x), p(i), z) \llbracket i \succ 0 \rrbracket\}$ . To prove termination of  $R_2$ , it suffices to show that the DP problem  $\{(1), (3), (6), (15) \mid u^\sharp(x, i, z) \rightarrow u^\sharp(p(x), p(i), z) \llbracket i \succ 0 \rrbracket\}$  is finite. Let  $\mathcal{H}$  be a tree homomorphism such that  $\mathcal{H}(f^\sharp) = \mathcal{H}(u^\sharp) = x_1$  as well as Example 26. For the DP problem  $\{(1), (3), (6), (15)\}$ , the DP processor in Theorem 25 outputs  $\{\{(1), (3), (6), (15)\}\}$  since the sequence “(1), (15)” cannot guarantee the existence of a lower bound of infinite chains containing infinitely many “(1), (15)” and thus  $S_{\text{bound}}$  does not include (1). However, any of the sequences “(1), (3)”, “(1), (6)”, “(3), (15)”, “(15), (3)” and “(15), (6)” guarantees the existence of lower bounds for infinite chains containing infinitely many of the sequence. Thus, it is enough to consider infinite chains such that the sequences do not appear in the infinite chain. All of such infinite chains can be obtained from graphs such that edges which correspond to the sequence are eliminated from  $DG(\{(1), (3), (6), (15)\}, R_2)$ . Therefore, it suffices to show finiteness of the DP problems  $\{(3)\}$  and  $\{(15)\}$  that are SCCs of the graph obtained from  $DG(\{(1), (3), (6), (15)\}, R_2)$  by eliminating edges ((1), (3)), ((1), (6)), ((3), (15)), ((15), (3)) and ((15), (6)) (Fig. 4).

Here, we define notations related to graphs. Let  $G$  be a graph  $(V, E)$ . The *vertex* and *edge* sets of  $G$  are denoted by  $V(G)$  and  $E(G)$ , respectively, i.e.,  $V = V(G)$  and  $E = E(G)$ . Let  $V' \subseteq V$ .  $G \setminus_{V'}$  denotes the graph obtained from  $G$  by removing  $V'$  from the vertex set and by removing edges related to  $V'$  from the edge set, i.e.,  $G \setminus_{V'} = (V(G) \setminus V', E(G) \setminus (V(G) \times V' \cup V' \times V(G)))$ . The set



**Fig. 4.**  $DG(\{(1), (3), (6), (15)\}, R_2)$  (left), and the graph (right) obtained from the left one by eliminating edges  $((1), (3))$ ,  $((1), (6))$ ,  $((3), (15))$ ,  $((15), (3))$  and  $((15), (6))$

of subgraphs of  $G$  is denoted by  $\text{Subg}(G)$ , i.e.,  $\text{Subg}(G) = \{(V', E') \mid V' \subseteq V(G), E' \subseteq E(G) \cap (V' \times V')\}$ . A graph  $G_1$  in  $\text{Subg}(G)$  is called *maximal in*  $X \subseteq \text{Subg}(G)$  if  $G_1 = G_2$  whenever  $G_1$  is a subgraph of  $G_2$  for all  $G_2 \in X$ .

We first extend chains to subgraphs of dependency graphs, and then adapt the termination criterion (Theorems 6 and 7) to the extended chains.

**Definition 28.** Let  $R$  be a constrained TRS and  $G$  be a subgraph of  $DG(R)$ . A  $V(G)$ -chain  $s_1 \rightarrow t_1 \llbracket \phi_1 \rrbracket, s_2 \rightarrow t_2 \llbracket \phi_2 \rrbracket, \dots$  is called a  $G$ -chain (of  $R$ ) if  $(s_i \rightarrow t_i \llbracket \phi_i \rrbracket, s_{i+1} \rightarrow t_{i+1} \llbracket \phi_{i+1} \rrbracket) \in E(G)$  for all  $i \geq 1$ . The  $G$ -chain is called *minimal* if it is a minimal  $V(G)$ -chain. Moreover, the infinite  $G$ -chain is called  $G$ -innumerable if it is a  $V(G)$ -innumerable chain and for every edge  $(s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket) \in E(G)$ , the subsequence  $s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket$  appears infinitely many times in the chain.

**Theorem 29.** A constrained TRS  $R$  is terminating iff there exists no infinite minimal  $DG(R)$ -chain.

**Theorem 30.** Let  $R$  be a constrained TRS,  $G$  be a subgraph of  $DG(R)$  and  $X \subseteq \text{Subg}(G)$  such that there exists no  $G'$ -innumerable chain for any  $G' \in X$ . There exists no infinite (minimal)  $G$ -chain iff there exists no infinite (minimal)  $G'$ -chain for any maximal graph  $G'$  in  $\text{Subg}(G) \setminus X$ .

The proofs of Theorem 29 and 30 can be found in [29].

Next, we extend the DP framework to the one such that DP processors handle subgraphs of dependency graphs. Let  $R$  be a constrained TRS. A subgraph of  $DG(R)$  is called a *graph-based DP problem* (of  $R$ ) (GDP problem, for short). A GDP problem  $G$  is called *finite* if there is no infinite minimal  $G$ -chain. A GDP problem  $G$  is called *infinite* if it is not finite or  $R$  is not terminating. A *graph-handling DP processor* (GDP processor, for short) is a function  $\text{Proc}$  which takes a GDP problem as input and returns a finite set of GDP problems. A GDP processor  $\text{Proc}$  is called *sound* if for any GDP problem  $G$ , the problem  $G$  is finite whenever all GDP problems in  $\text{Proc}(G)$  are finite.  $\text{Proc}$  is called *complete* if for any GDP problem  $G$ , the problem  $G$  is infinite whenever  $\text{Proc}(G)$  contains infinite GDP problems. Therefore, a termination proof starts with the initial GDP problem  $DG(R)$  and applies sound GDP processors until an empty graph of GDP problems is obtained.

**Theorem 31.** A constrained TRS  $R$  is terminating iff the GDP problem  $DG(R)$  is finite. Moreover,  $R$  is not terminating iff the GDP problem  $DG(R)$  is infinite.

*Proof.* This theorem follows from Theorem 29. □

As a tool to show soundness and completeness of GDP processors, we extend Theorem 19 as follows.

**Theorem 32.** *Let  $R$  be a constrained TRS. A GDP processor  $Proc$  is sound and complete if for any  $G$  which is a subgraph of  $DG(R)$ , there exists no  $G'$ -innumerable chain for any  $G'$  such that  $V(G') \setminus V(G'') \neq \emptyset$  or  $E(G') \setminus E(G'') \neq \emptyset$  for all  $G'' \in Proc(G)$ .*

*Proof.* The proof is similar to that of Theorem 19. □

Next, we show how to adapt the existing DP processors to GDP problems.

**Theorem 33.** *Let  $R$  be a constrained TRS,  $G$  be a subgraph of  $DG(R)$  and  $Proc_p$  be a DP processor. The following GDP processor  $Proc_g$  is sound if  $Proc_p$  is sound:*

$$Proc_g(G) = \begin{cases} \{DG(S, R) \mid S \in Proc_p(V(G))\} & \text{if } V(G) \notin Proc_p(V(G)) \\ \{G\} & \text{otherwise} \end{cases} \quad (1)$$

*Suppose that for any infinite (minimal)  $V(G)$ -chain, there exists some graph  $S \in Proc_p(V(G))$  such that there exists an infinite (minimal)  $S$ -chain as a postfix chain of the  $V(G)$ -chain. Then, the following GDP processor  $Proc_g$  is sound and complete if  $Proc_p$  is sound and complete:*

$$Proc_g(G) = \begin{cases} \{(S, E(G) \cap S \times S) \mid S \in Proc_p(V(G))\} & \text{if } V(G) \notin Proc_p(V(G)) \\ \{G\} & \text{otherwise} \end{cases} \quad (2)$$

The proof can be found in [29]. The first adaptation (1) in Theorem 33 is naive and the processor re-computes dependency graphs. Due to the re-computation, all the eliminated edges are restored and thus the application of the processor is sometimes meaningless. Moreover, almost all the existing practical and *non-transformational*<sup>5</sup> DP processors (e.g., processors based on dependency graphs, *reduction pairs* [25] and the subterm criterion [22]) satisfy the assumption that is required to employ (2) in Theorem 33. For these reasons, we prefer (2) in Theorem 33 to (1).

Thanks to Theorem 33, the DP processor in Theorem 20 can be adapted to GDP problems. However, the adapted processor is not always useful since for a GDP problem  $G$  such that if  $DG(V(G), R)$  is an SCC then it outputs not a set of SCCs of  $G$  but  $\{G\}$ . For this reason, we extend the DP processor in Theorem 20 to a GDP one, without the construction in Theorem 33.

**Theorem 34.** *Let  $R$  be a constrained TRS and  $G$  be a subgraph of  $DG(R)$ . Then, the GDP processor, which takes a GDP problem  $G$  and outputs SCCs of  $G$ , is sound and complete.*

*Proof.* Let  $X = \{G' \mid G' \text{ is not a subgraph of for any SCC of } G\}$ . For all  $G' \in X$ , there exists no  $G'$ -innumerable chain since  $G'$  is not a strongly connected graph. Thus, the processor is sound and complete it follows from Theorem 32. □

<sup>5</sup> A DP processor is called *non-transformational* if it returns sets of dependency pairs.



Finally, we extend the DP processor in Theorem 25 to a GDP one that eliminates none of nodes but some of edges. For a dependency pair, the processor in Theorem 25 does not always succeed in detecting a lower bound or in ordering by  $>$  with all the dependency pairs following after the focusing dependency pair, while it is possible to do for some of the accident dependency pairs. As stated in Example 27, eliminating the edges from the focusing dependency pair to the accident dependency pairs is sound for proving termination.

**Theorem 35.** *Let  $R$  be a locally sound constrained TRS over  $(\mathcal{F}, \mathcal{G}_{\mathbb{Z}}, \mathcal{P}_{\mathbb{Z}}, \mathcal{V}, \mathcal{M}_{\mathbb{Z}})$ ,  $G$  be a subgraph of  $DG(R)$ , and  $\mathcal{H}$  be a tree homomorphism from  $\mathcal{D}_R^\sharp$  to  $T(\mathcal{F} \cup \mathcal{G}_{\mathbb{Z}}, \mathcal{V})$  such that  $\mathcal{H}(s), \mathcal{H}(t) \in T(\mathcal{G}, \mathcal{V})$  for any  $s \rightarrow t \llbracket \phi \rrbracket \in V(G)$ . Moreover, let*

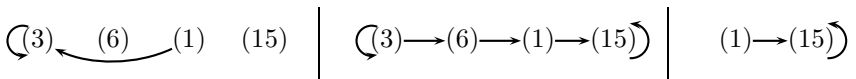
- $E_{\succ} = \{(s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket) \mid \text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \text{Var}(\mathcal{H}(s))) \Rightarrow \mathcal{H}(s) \succ \mathcal{H}(t) \text{ is valid w.r.t. } \mathcal{M}_{\mathbb{Z}}\}$ ,
- $E_{\succsim} = \{(s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket) \mid \text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \text{Var}(\mathcal{H}(s))) \Rightarrow \mathcal{H}(s) \succ \mathcal{H}(t) \vee \mathcal{H}(s) \simeq \mathcal{H}(t) \text{ is valid w.r.t. } \mathcal{M}_{\mathbb{Z}}\}$ ,
- $E_{\text{bound}} = \{(s \rightarrow t \llbracket \phi \rrbracket, u \rightarrow v \llbracket \psi \rrbracket) \mid \exists s_{\text{bound}} \in T(\mathcal{G}_{\mathbb{Z}}), \text{chain}(s \rightarrow t \llbracket \phi \rrbracket, u' \rightarrow v' \llbracket \psi' \rrbracket, \text{Var}(\mathcal{H}(s))) \Rightarrow \mathcal{H}(s) \succ s_{\text{bound}} \text{ is valid w.r.t. } \mathcal{M}_{\mathbb{Z}}\}$ ,
- $S_{\text{filter}} = \{s \rightarrow t \llbracket \phi \rrbracket \mid \text{Var}(\mathcal{H}(s)) \subseteq \text{fv}(\phi)\}$ , and
- $S_{\text{prsrv}} = \{s \rightarrow t \llbracket \phi \rrbracket \mid \text{Var}(\mathcal{H}(t)) \subseteq \text{fv}(\phi) \cup \text{Var}(\mathcal{H}(s))\}$ ,

where  $u' \rightarrow v' \llbracket \psi' \rrbracket$  is a renamed variant of  $u \rightarrow v \llbracket \psi \rrbracket$  such that  $\text{Var}(s) \cap \text{Var}(u') = \emptyset$ . If none of  $E(G) \cap E_{\succ}$ ,  $E(G) \cap E_{\text{bound}}$  and  $V(G) \cap S_{\text{filter}}$  is empty,  $E(G) \subseteq E_{\succsim}$  and  $V(G) \subseteq S_{\text{prsrv}}$ , then there exists no  $G$ -innumerable chain, i.e., the following processor  $\text{Proc}$  is sound and complete:

$$\text{Proc}(G) = \begin{cases} \{(V(G), E(G) \setminus E_{\succ}), (V(G), E(G) \setminus E_{\text{bound}}), (G \setminus S_{\text{filter}})\} & \text{if } V(G) \subseteq S_{\text{prsrv}} \\ & \text{and } E(G) \subseteq E_{\succsim} \\ \{G\} & \text{otherwise} \end{cases}$$

*Proof.* The proof is similar to that of Theorem 25. □

*Example 36.* Consider the constrained TRS  $R_2$  in Example 27 again. Let  $\mathcal{H}$  be a tree homomorphism such that  $\mathcal{H}(f^\sharp) = \mathcal{H}(u^\sharp) = x_1$ . For the GDP problem  $DG(\{(1), (3), (6), (15)\}, R_2)$ , the GDP processor in Theorem 35 outputs the three GDP problems illustrated in Fig. 5. By using the GDP processors in Theorems 34 and 35, it is possible to show that these three GDP problems are finite and hence  $DG(\{(1), (3), (6), (15)\}, R_2)$  is finite. Therefore,  $R_2$  is terminating.



**Fig. 5.** the GDP problems obtained from  $DG(\{(1), (3), (6), (15)\}, R_2)$  by applying the GDP processor in Theorem 35



## 7 Conclusion

In this paper, we have extended both the termination criterion w.r.t. dependency pairs and the DP framework to constrained TRSs, and have proposed a DP processor, which is a simplified combination of the lower bound detection and the polynomial interpretation method over integers. Then we have extended the DP framework for constrained TRSs to the GDP framework and have shown how to adapt the existing DP processors to the GDP framework. As future work, we will adapt the GDP framework for constrained TRSs to transformational processors such as the *argument filtering* processor [19], and will implement the GDP framework. Furthermore, we will improve the GDP processors, e.g., the one in Theorem 35.

## References

1. Armando, A., Rusinowitch, M., Stratulat, S.: Incorporating decision procedures in implicit induction. *J. Symb. Comput.* 34(4), 241–258 (2002)
2. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.* 236(1–2), 133–178 (2000)
3. Baader, F., Nipkow, T.: *Term Rewriting and All That*. Cambridge University Press, United Kingdom (1998)
4. Borralleras, C., Rubio, A.: Orderings and constraints: Theory and practice of proving termination. In: Comon-Lundh, H., Kirchner, C., Kirchner, H. (eds.) *Jouannaud Festschrift. LNCS*, vol. 4600, pp. 28–43. Springer, Heidelberg (2007)
5. Bouhoula, A., Rusinowitch, M.: Implicit induction in conditional theories. *Journal of Automated Reasoning* 14(2), 189–235 (1995)
6. Bouhoula, A., Jacquemard, F.: Automated induction for complex data structures. In: *CoRR*, abs/0811.4720 (2008)
7. Bouhoula, A., Jacquemard, F.: Automated induction with constrained tree automata. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) *IJCAR 2008. LNCS (LNAI)*, vol. 5195, pp. 539–554. Springer, Heidelberg (2008)
8. Comon, H.: Completion of rewrite systems with membership constraints. In: Kuich, W. (ed.) *ICALP 1992. LNCS*, vol. 623, pp. 392–403. Springer, Heidelberg (1992)
9. Comon, H.: Completion of rewrite systems with membership constraints. part I: Deduction rules. *J. Symb. Comput.* 25(4), 397–419 (1998)
10. Comon, H.: Completion of rewrite systems with membership constraints. part II: Constraint solving. *J. Symb. Comput.* 25(4), 421–453 (1998)
11. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: *Tree automata techniques and applications* (2007), <http://www.grappa.univ-lille3.fr/tata> (release October 12, 2007)
12. Falke, S., Kapur, D.: Inductive decidability using implicit induction. In: Hermann, M., Voronkov, A. (eds.) *LPAR 2006. LNCS (LNAI)*, vol. 4246, pp. 45–59. Springer, Heidelberg (2006)
13. Falke, S., Kapur, D.: Dependency pairs for rewriting with built-in numbers and semantic data structures. In: Voronkov, A. (ed.) *RTA 2008. LNCS*, vol. 5117, pp. 94–109. Springer, Heidelberg (2008)
14. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt, R.A. (ed.) *CADE-22. LNCS*, vol. 5663, pp. 277–293. Springer, Heidelberg (2009)

15. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: Maximal termination. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 110–125. Springer, Heidelberg (2008)
16. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving termination of integer term rewriting. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 32–47. Springer, Heidelberg (2009)
17. Furuichi, Y., Nishida, N., Sakai, M., Kusakari, K., Sakabe, T.: Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. IPSJ Trans. on Prog. 1(2), 100–121 (2008) (in Japanese)
18. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJ-CAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
19. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
20. Giesl, J., Thiemann, R., Swiderski, S., Schneider-Kamp, P.: Proving termination by bounded increase. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 443–459. Springer, Heidelberg (2007)
21. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 32–46. Springer, Heidelberg (2003)
22. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. Inf. Comput. 205(4), 474–511 (2007)
23. Hoot, C.: Completion for constrained term rewriting systems. In: Rusinowitch, M., Remy, J.-L. (eds.) CTRS 1992. LNCS, vol. 656, pp. 408–423. Springer, Heidelberg (1993)
24. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, Cambridge (2000)
25. Kusakari, K., Nakamura, M., Toyama, Y.: Argument filtering transformation. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 47–61. Springer, Heidelberg (1999)
26. Lynch, C., Snyder, W.: Redundancy criteria for constrained completion. Theor. Comput. Sci. 142(2), 141–177 (1995)
27. Nishida, N., Sakai, M., Hattori, T.: On Disproving Termination of Constrained Term Rewriting Systems. In: Proc. of WST 2010, 5 pages (2010)
28. Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer, Heidelberg (2002)
29. Sakata, T., Nishida, N., Sakabe, T.: On proving termination of constrained term rewriting systems by eliminating edges from dependency graphs. The Full Version of this Paper, <http://www.trs.cm.is.nagoya-u.ac.jp/~nishida/wflp11/>
30. Sakata, T., Nishida, N., Sakabe, T., Sakai, M., Kusakari, K.: Rewriting induction for constrained term rewriting systems. IPSJ Trans. on Prog. 2(2), 80–96 (2009) (in Japanese)
31. Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. PhD thesis, RWTH Aachen University, Germany (October 2007)
32. Toyama, Y.: Confluent term rewriting systems with membership conditions. In: Kaplan, S., Jouannaud, J.-P. (eds.) CTRS 1987. LNCS, vol. 308, pp. 228–241. Springer, Heidelberg (1988)
33. Yamada, J.: Confluence of terminating membership conditional TRS. In: Rusinowitch, M., Remy, J.-L. (eds.) CTRS 1992. LNCS, vol. 656, pp. 378–392. Springer, Heidelberg (1993)

# Author Index

Almendros-Jiménez, Jesus M.	35	Nishida, Naoki	138
Antoy, Sergio	19	Peemöller, Björn	1
Braßel, Bernd	1	Peña, Ricardo	52
Caballero, Rafael	35	Reck, Fabian	1
Delgado-Muñoz, Agustin D.	52	Rodrigues, Vítor	86
Florido, Mário	86	Sáenz-Pérez, Fernando	35
García-Ruiz, Yolando	35	Sakabe, Toshiki	138
Hanus, Michael	1, 19	Sakata, Tsubasa	138
Kuchen, Herbert	122	Samulowitz, Horst	68
Majchrzak, Tim A.	122	Schrijvers, Tom	68
Melo de Sousa, Simão	86	Stuckey, Peter	68
		Tack, Guido	68
		Wuille, Pieter	68
		Zinn, Claus	104