Kostis Sagonas (Ed.)

LNCS 7752

Practical Aspects of Declarative Languages

15th International Symposium, PADL 2013
Rome, Italy, January 2013
Proceedings

Springer

Kostis Sagonas (Ed.)

Practical Aspects of Declarative Languages

15th International Symposium, PADL 2013
Rome, Italy, January 21-22, 2013
Proceedings

Springer

Volume Editor

Kostis Sagonas
Uppsala University
Department of Information Technology
Box 337, 751 05 Uppsala, Sweden
E-mail: kostis@it.uu.se

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-45283-3 e-ISBN 978-3-642-45284-0
DOI 10.1007/978-3-642-45284-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013954880

CR Subject Classification (1998): D.3, D.1, F.3, D.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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)

Preface

Declarative languages build on sound theoretical bases to provide attractive frameworks for application development. These languages have been successfully applied to many different real-world situations, ranging from database management, to active networks, to software engineering, to decision support systems.

New developments in theory and implementation have opened up new application areas. At the same time, applications of declarative languages to novel problems raise numerous interesting research issues. Well-known questions include designing for scalability, language extensions for application deployment, and programming environments. Thus, applications drive the progress in the theory and implementation of declarative systems, and benefit from this progress as well.

PADL is a forum for researchers and practitioners to present original work emphasizing novel applications and implementation techniques for all forms of declarative concepts, including functional, logic, constraints, etc. This volume contains the papers presented at PADL 2013: the 15th International Symposium on Practical Aspects of Declarative Languages held during January 20–21, 2013, in Rome.

There were 33 submissions. Each submission was reviewed by at least three, and on average four, Program Committee members. The committee decided to accept 17 papers. The program also includes one invited talk by Tom Schrijvers.

PADL 2013 was co-located with ACM's 40th Symposium on Principles of Programming Languages. Previous PADL symposia were held in Philadelphia, Pennsylvania, USA (2012), Austin, Texas, USA (2011), Madrid, Spain (2010), Savannah, Georgia, USA (2009), San Francisco, California, USA (2008), Nice, France (2007), Charleston, South Carolina, USA (2006), Long Beach, California, USA (2005), Dallas, Texas, USA (2004), New Orleans, Louisiana, USA (2003), Portland, Oregon, USA (2002), Las Vegas, Nevada, USA (2001), Boston, Massachusetts, USA (2000), and San Antonio, Texas, USA (1999).

PADL 2013 was sponsored by the Association for Logic Programming and organized in co-operation with ACM SIGPLAN. Thanks to Matt Might, the POPL workshop chair, for his help and guidance. Thanks also to Gopal Gupta, president of the Association for Logic Programming, for his help and support. Lastly, thanks to the EasyChair system that made managing paper submissions, paper reviewing, and preparation of the proceedings a breeze.

October 2013 Kostis Sagonas

Organization

Program Committee

Elvira Albert	Complutense University of Madrid, Spain
Adam Chlipala	Massachusetts Institute of Technology, USA
Bart Demoen	KU Leuven, Belgium
Maurizio Gabbrielli	University of Bologna, Italy
Maria Garcia de La Banda	Monash University, Australia
Jurriaan Hage	Utrecht University, The Netherlands
Kevin Hammond	University of St. Andrews, UK
Ralf Hinze	University of Oxford, UK
Jacob Howe	City University London, UK
Joxan Jaffar	National University of Singapore, Singapore
Gabriele Keller	The University of New South Wales, Australia
Naoki Kobayashi	University of Tokyo, Japan
Kim Nguyen	LRI, Université Paris-Sud, France
Norman Ramsey	Tufts University, USA
Kostis Sagonas	Uppsala University, Sweden and NTUA, Greece
Vítor Santos Costa	University of Porto, Portugal
Olin Shivers	Northeastern University, USA
Terrance Swift	Universidade Nova de Lisboa, Portugal and Johns Hopkins University, USA
Dimitrios Vytiniotis	Microsoft Research, UK

Additional Reviewers

Amadini, Roberto
Arenas, Puri
Auzende, Odette
Brady, Edwin
Bruynooghe, Maurice
Calejo, Miguel
Cheung, Kwok
Chin, Wei-Ngan
Correas Fernández, Jesús
Dijkstra, Atze
Duck, Gregory
Falaschi, Moreno
Feng, Yuzhang
Genaim, Samir
Greenaway, David
Harper, Thomas
James, Daniel
Kameya, Yoshitaka
Khoo, Siau-Cheng
King, Andy
Ko, Yousun
Lanese, Ivan
Leung, Ho-Fung
Magalhães, José Pedro
Majchrzak, Tim A.
Martin-Martin, Enrique
Mauro, Jacopo
Mendez-Lojo, Mario
Meo, Maria Chiara
Meyer, Bernd

Pinto, Alexandre Miguel	Tack, Guido
Santosa, Andrew	Vaz, David
Sato, Taisuke	Wilson, Walter
Shepherd, John	Wu, Nicolas
Stuckey, Peter	Zhou, Wenchao
Sánchez-Hernández, Jaime	

Invited Talk
(Abstract)

Zipping Trees Across the Border

From Functional Specification to Logic Programming Implementation

Tom Schrijvers

Department of Applied Mathematics and Computer Science
Ghent University
tom.schrijvers@ugent.be

Abstract. The story starts in Haskell land: A humble monoidal operator for `zipping` trees embarks on an epic journey. Armed with monads it rides into Prolog land and wears down a meta-interpreter with delimited continuations, thereby freeing the Prolog folks from the yoke of depth-first search. They lived happily ever after with modular declarative search heuristics.

Prolog has a depth-first procedural semantics. Unfortunately, this procedural semantics is ineffective for many programs. Instead, to compute useful solutions, it is necessary to modify the search method that explores the alternative execution branches with various kinds of heuristics. For instance, the code on the left implements a typical Constraint Logic Programming labeling predicate, while the code on the right adds a depth-bounded search heuristic to it.

```
label([]).
label([Var|Vars]) :-
  ( var(Var) ->
      fd_inf(Var,Value),
      ( Var #= Value,
        label(Vars)
      ; Var #\= Value,
        label([Var|Vars])
      )
  ;
      label(Vars)
  ).
```

```
label([],_).
label([Var|Vars],D) :-
  ( var(Var) ->
      D > 0,
      ND is D - 1,
      fd_inf(Var,Value),
      ( Var #= Value,
        label(Vars,ND)
      ; Var #\= Value,
        label([Var|Vars],ND)
      )
  ;
      label(Vars,D)
  ).
```

Manually adapting Prolog programs to incorporate search heuristics obviously has many disadvantages: it is labor intensive, error prone, tedious and the bookkeeping required for adaptations to non-trivial programs quickly exceeds the mental capacities of programmers.

Some Prolog systems, like Ciao [1], offer a limited number of heuristics through automatic program transformation of programmer annotated predicates. The drawback is that a new heuristic requires a new full-blown program transformation. The Tor library [2], available in SWI-Prolog [3], supports the development of new search heuristics; with a hookable disjunction these can be applied in a very flexible, compositional and convenient way to any programs. However, search heuristics have to be specified in a very operational manner. This can be an impediment in the development of new heuristics.

In this work we provide a cleaner alternative for specifying new search heuristics by taking a page from the book of functional programming.

We start from a simplified functional specification in Haskell: the `zipTree` operation on trees that is similar to the `zip` and `zipWith` family of Haskell list functions. The idea is that one tree (the heuristic) bends the other one (the actual search) into the desired shape. Then we generalize our trees by interleaving them with (monadic) effects in the style of Atkey et al. [4] to more faithfully characterize search in Prolog. Next we ship the Haskell specification across the language border to construct a Prolog meta-interpreter. Finally, we derive a direct library-based Prolog implementation by transforming the interpreter into continuation passing style, specializing it and adding support for delimited continuations to the WAM.

The end result is that we can write the depth-bounded search heuristic as a simple declarative predicate in Prolog:

```
dbs(Depth) :-
  Depth > 0,
  NDepth is Depth - 1,
  ( dbs(NDepth)
  ; dbs(NDepth)
  ).
```

and neatly apply it to `lable/1` as follows:

```
label(Vars,Depth) :-
  zipTree(dbs(Depth),label(Vars)).
```

Acknowledgments. This work in progress is conducted jointly with Bart Demoen en Benoit Desouter.

References

1. Hermenegildo, M.V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J.F., Puebla, G.: An overview of Ciao and its design philosophy. Theory and Practice of Logic Programming **12** (2012) 219–252
2. Schrijvers, T., Triska, M., Demoen, B.: Tor: extensible search with hookable disjunction. In King, A., ed.: Principles and Practice of Declarative Programming, 14th International symposium, Proceedings, ACM SIGPLAN (2012) 12
3. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12** (2012) 67–96
4. Atkey, R., Johann, P., Ghani, N., Jacobs, B.: Interleaving data and effects. Submitted for Publication (2012)

Table of Contents

A Library for Declarative Resolution-Independent 2D Graphics

Paul Klint and Atze van der Ploeg

Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{paulk,ploeg}@cwi.nl

Abstract. The design of most 2D graphics frameworks has been guided by what the computer can draw efficiently, instead of by how graphics can best be expressed and composed. As a result, such frameworks restrict expressivity by providing a limited set of shape primitives, a limited set of textures and only affine transformations. For example, non-affine transformations can only be added by invasive modification or complex tricks rather than by simple composition. More general frameworks exist, but they make it harder to describe and analyze shapes. We present a new declarative approach to resolution-independent 2D graphics that generalizes and simplifies the functionality of traditional frameworks, while preserving their efficiency. As a real-world example, we show the implementation of a form of focus+context lenses that gives better image quality and better performance than the state-of-the-art solution at a fraction of the code. Our approach can serve as a versatile foundation for the creation of advanced graphics and higher level frameworks.

Keywords: Declarative Graphics, Design, Resolution-Independence, Optimization, Focus+context lenses.

1 Introduction

The design of traditional 2D graphics frameworks, such as Java2D[1] and Processing[2], has been guided by what the computer can draw efficiently, instead of by how graphics can best be expressed and composed. This hinders the ease of programming 2D graphics, since it requires the programmer to express his ideas using the limited vocabulary that has emerged as a result of the focus on procedural optimization of such frameworks.

Suppose we have programmed a visualization in such a traditional framework and we now want to add a focus+context lens, such as the one shown in Figure 1. Since only affine transformations (that take parallel lines to parallel lines) are supported, we cannot add this transformation in a compositional way: it requires trickery or invasive modification.

Instead of worrying about such low-level details, it is desirable to program 2D graphics in a declarative way that is general, simple, expressive, composable and resolution-independent while still being efficient. Previous research on declarative graphics has yielded many elegant approaches to 2D graphics, but none of

[1] `http://docs.oracle.com/javase/6/docs/technotes/guides/2d/`

[2] `http://processing.org`

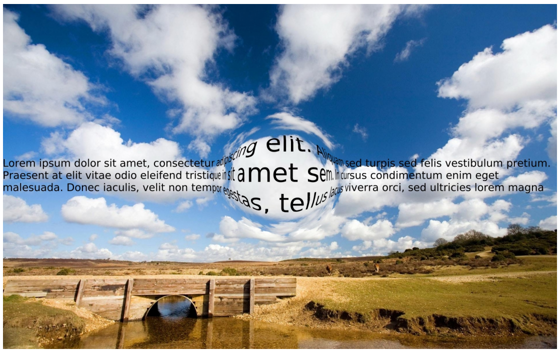

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam sed turpis sed felis vestibulum pretium.
Praesent at elit vitae odio eleifend tristique in sit amet sem. In cursus condimentum enim eget
malesuada. Donec iaculis, velit non tempor egestas, tellus lacus viverra orci, sed ultricies lorem magna

Fig. 1. An example focus+context lens (zoomfactor = 2.5)

these exhibit all these traits. This not only restricts direct graphics programming, but it also hinders the creation of higher-level frameworks. For example, during our efforts on the Rascal figure library[1], a high-level framework for software visualization, we noticed that our design was influenced by the limitations of the procedural framework used and hence could not grow further in terms of expressiveness and compositionality.

We present a new declarative approach that generalizes and simplifies the functionality of traditional 2D graphics frameworks, while preserving their efficiency. This is achieved by a very effective mapping of our approach to an existing 2D graphics framework (which we will call the *graphics host*). Our approach allows more expressive freedom and can hence serve as a more versatile foundation for advanced 2D graphics and higher-level frameworks. It is available as a library called *Deform*[3] for Scala. Our contributions are:

- The motivation (Section 2) and design (Section 3) of a small, simple and powerful framework for resolution-independent 2D graphics that enables composability and expressiveness.
- A way to implement and optimize this framework (Section 4) by mapping it to a readily-available, highly optimized graphics host. This includes optimizations to speed up this mapping and a way to support clipping so that large scenes can be rendered in real-time.
- An implementation of focus+context lenses that is faster and gives better image quality than the state-of-the-art approach (Section 5). This also acts as a validation of our work.

We discuss open questions in Section 6 and conclude in Section 7.

2 Exploring the Design Space

We now discuss design choices for declarative 2D graphics frameworks and to guide our choices, we use the following design goals:

[3] `https://github.com/cwi-swat/deform`

– *Simplicity*: The programmer should not be overwhelmed by concepts and functions described in inch-thick manuals.
– *Expressivity*: Arbitrary graphics can be expressed in a *natural* way, without the need to encode them in lower-level concepts.
– *Composability*: Graphics can be composed and transformed in general ways.
– *Resolution-independence*: Graphics can be expressed independent of resolution, so that they can be rendered at any level of detail.
– *Analyzability*: The concrete geometry of a shape can be obtained, for example as a list of lines and Bézier curves, so that we can define functions that act on this information to create derived graphics.
– *Optimizability*: Efficient algorithms for 2D graphics can be re-used.

Our analysis now focuses on how to represent *shapes*, *textures* and *transformations*, in the way that has the best fit with our design goals.

2.1 Shapes

Most frameworks offer a fixed set of geometric constructs, such as lines, Bézier curves and circle segments, that can be used to describe the *border* of shapes. For example, a regular polygon with k vertices can be expressed as follows:

$$\begin{aligned}&\mathit{regpolyg}(k) = [\mathit{line}(\mathit{onCircle}(i\times p), \mathit{onCircle}((i+1)\times p)) \,|\, i \leftarrow [0\ldots k-1]]\\&\qquad\mathbf{where}\ \mathit{onCircle}(x) = \langle \sin(x), \cos(x)\rangle,\quad p = (1/k)\times 2\times\pi\end{aligned}$$

Here $\langle x, y\rangle$ denotes a point in $\mathbb{R}^2$. A downside of this approach is that shapes that are not compositions of such geometric constructs, such as sine waves, cannot be expressed. Instead, they have to be approximated *when specifying the shape*, which does not give a resolution-independent description of the shape.

A second approach is to describe the border of a shape as a *parametric* curve: a function from $\mathbb{R}$ to $\mathbb{R}^2$. For example, the border of the unit circle can be described by $c(t) = \langle \sin(t\times 2\times\pi), \cos(t\times 2\times\pi)\rangle$ on the interval $[0, 1]$. This can be seen as a generalization of using a fixed set of geometric constructs: each geometric construct can be described by a parametric curve and hence a combination of geometric constructs gives rise to a piecewise defined function. For this reason the expression of a regular polygon with k vertices is exactly the same as when using a fixed set of geometric constructs. Although a parametric description does not immediately give an analyzable description of the shape, we can sample the (resolution-independent) function to obtain such a description.

The third and final approach is to describe a shape *implicitly*: as a function that given a point in $\mathbb{R}^2$ tells us whether the point is inside the shape or not. For example, the implicit representation of the unit circle is $c(p) = |p| \leq 1$, where $|p|$ denotes the Euclidian norm. A downside of this approach is that it is often hard to encode a shape in this way. For example, as noted in [2], it requires an arcane insight to understand that the following also represents a regular polygon with k vertices.

$$\begin{aligned}&\mathit{regpolyg}\ (k, \langle x, y\rangle) = (x-j)\times(\sin(q+p)-i) - (\cos(q+p)-j)\times(y-i) \leq 0\\&\qquad\mathbf{where}\, p = 2\times\pi/k,\quad q = p\times\lfloor \mathit{atan2}(y, x)/p\rfloor,\quad i = \sin(q),\quad j = \cos(q)\end{aligned}$$

It is also hard to analyze a shape that is described in this way, since we do not have a representation of the border of the shape.

If we could automatically switch between the parametric and implicit representations we would not have to make a choice between them. However, transforming a parametric representation into an implicit one or vice-versa is non-trivial, especially when the functions are not limited to a certain class. In fact, these are well-known and thoroughly studied problems [3]. In general, exact conversion is possible for certain classes of functions [4], while other classes of functions require approximate techniques [5]. Since the implicit representation makes it hard to express and analyze shapes, and since it is hard and computationally expensive to automate the conversion between the two representations we have chosen to describe shapes parametrically.

2.2 Textures

Most frameworks offer a fixed set of textures, such as fill colors, images and gradients. Another approach is allow arbitrary textures by specifying the colors of its pixels, but this is not a resolution independent approach. A general, resolution independent way to describe a texture, and the one that we adopt, is by a function that given a point returns the color of the texture at that point [6,2]. Notice that this way of expressing textures bears resemblance to implicitly defined shapes: implicitly defined shapes are functions of type $\mathbb{R}^2 \to \mathit{Boolean}$, whereas such textures are functions of type $\mathbb{R}^2 \to \mathit{Color}$.

2.3 Transformations

Typically, graphics frameworks offer only affine transformations, such as translation, rotation and scaling. Although these transformations cover many use cases, they preclude a whole range of interesting transformations, such as focus+context lenses. A more expressive model is to describe transformations simply as a function from $\mathbb{R}^2$ to $\mathbb{R}^2$.

Parametrically described shapes then require the *forward* transformation, while textures and implicitly defined shapes require the *inverse* transformation. For example, to translate a parametrically defined shape to the right, we define a function that given a parameter first gets the corresponding point on the border of the shape and then applies the forward transformation to that point, which moves the point to the right. To translate a texture to the right, we define a function that given a point first applies the inverse transformation, which moves the point to the left, and then queries the texture at that point. In the same fashion, the inverse transformation is also needed to transform implicitly defined shapes.

If we limit ourselves to affine transformations, obtaining both directions of a transformation is not a problem since such transformations are easily inverted. However, if we allow arbitrary transformations we need to either describe all shapes implicitly and use only the inverse transformation, making it harder to describe shapes, or describe shapes parametrically in which case we need *both*

Table 1. Design choices for graphics libraries

		Traditional	Func. image synthesis	Vertigo	Deform
Shapes	Fixed	●			
	Parametric			●	●
	Implicit		●		
Textures	Fixed/pixels	●			
	Function		●		●
Transformations	Affine	●			
	Function			●	●
	Function^{-1}		●		●

the forward transformation and the inverse transformation, making it harder to describe transformations. We conjecture that shapes are more likely to be application-specific than transformations, which can often be reused. Hence, we have chosen to represent shapes parametrically and require a definition of both directions for transformations.

2.4 Comparison

As a comparison, Table 1 lists the choices made by us and other frameworks. *Traditional* frameworks, like as Java2D, Processing and many others, limit the expressivity of the programmer by only providing support for the most common use cases. Many declarative graphics frameworks[4] make the same choices [7,8]. *Functional image synthesis* frameworks, such as Pan [6] and Clastic [2], are based on the notion that an image is simply a function from a point to a color. This allows the elegant definition of many interesting visual mathematical graphics but precludes real-life graphics, since the requirement of implicitly defined shapes makes hard to define complex shapes such as letters. *Vertigo* [9] is an elegant declarative framework for the geometric modeling of 3D shapes, without texturing. In *Deform* we have chosen a combination of design decisions that has not yet been explored: parametric shapes, textures as functions and general transformations. In the rest of this paper we show that this allows us to define a simple, general and resolution-independent framework which is applicable to real-life graphics.

3 Design

It is time to present our approach and illustrate its usage via examples. The basic unit of our framework is a *TexturedShape*, that describes a shape and the texture of its interior. An expression constituting a list of such textured shapes is first *created* using the constructors given in Table 2 and then *displayed* by a render function which interprets the constructors and produces an image. We will

[4] Unfortunately, space limitations do not allow a more extensive discussion.

Table 2. Constructors and functions. $[A]$ indicates a list of As.

Constructor	*Type*
path	$(\mathbb{R} \to \mathbb{R}^2) \to \mathit{Path}$
shape	$[\mathit{Path}] \to \mathit{Shape}$
analyze	$\forall A.\mathit{Path} \times (\mathit{ConcreteGeom} \to A) \to A$
color	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathit{Color}$
texture	$(\mathbb{R}^2 \to \mathit{Color}) \to \mathit{Texture}$
fill	$\mathit{Shape} \times \mathit{Texture} \to \mathit{TexturedShape}$
transformation	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \to \mathit{Transformation}$

now show how to express shapes, textures and transformations in this way. Our examples were programmed in Scala and then hand-transformed into a custom notation which should be easy to understand. The examples use the constructors in Table 2 and some library functions of Deform, both of which will be explained when used.

3.1 Shapes

The basis for describing shapes is the *path* constructor, which takes a parametric description of the border of the shape, a function of type $\mathbb{R} \to \mathbb{R}^2$. To allow omission of the domain of this function, it simply must be $[0, 1]$. The *shape* constructor can then be used to create a shape from a list of *closed* paths, paths of which the start and end points are the same. If one of the paths is not closed, then it does not define an area and a run-time error will be thrown. A point is then inside the shape if it is inside any of its closed paths.

As a basic example, consider a circle:

$$\mathit{circ} = \mathit{shape}([\mathit{path}(\lambda t \to \langle \sin(t \times 2 \times \pi), \cos(t \times 2 \times \pi)\rangle)])$$

The coordinate system of our framework is as follows: if the screen is square then the north west corner of the screen is $\langle -1, -1\rangle$ and the south east corner is $\langle 1, 1\rangle$. If the screen is non-square the range of the longest axis is adopted so that graphics maintain their aspect-ratio. An example of a more complex path is the spiral shown in Figure 2(a):

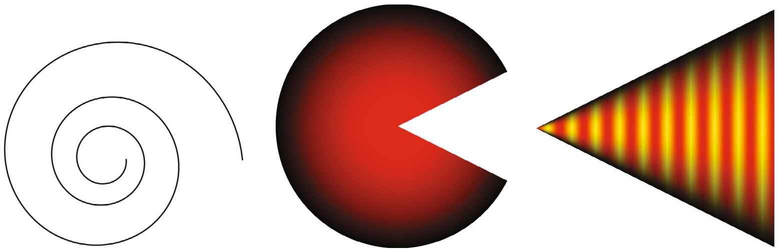
$$\begin{aligned} \mathit{spiral} = \mathit{path}(\lambda t \to \langle f \times \cos(s), f \times \sin(s)\rangle \\ \textbf{where } f = 1/50 \times e^{s/10}, \quad s = 6 \times \pi \times (1 + t) \end{aligned}$$

Paths themselves cannot be drawn as they do not define an area. Hence, to produce a drawing of this spiral we use the *stroke* library function to convert this path to a shape given the width of the "pen":

$$\mathit{stroke}(\mathit{spiral}, 1/200)$$

We do not have to explicitly define a parametric representation for each shape. Instead, we provide library functions that mimic the geometric constructs found in traditional libraries. For example, we can create a triangle as follows:

$$\begin{aligned} &\mathit{triangle} = \mathit{shape}([\mathit{join}([\mathit{line}(a,b), \mathit{line}(b,c), \mathit{line}(c,a)])]) \\ &\qquad\qquad \textbf{where}\, a = \langle 0,0\rangle,\ \ b = \langle 1, \tfrac{1}{2}\rangle,\ \ c = \langle 1, -\tfrac{1}{2}\rangle \end{aligned}$$



(a) A simple spiral (b) Circle with triangle subtracted (c) A filled triangle

Fig. 2. Basic examples

To define functions which act on the geometry of a path, such as the *stroke* function, we offer the *analyze* constructor which takes a path and a function transforming the concrete geometry of the path, namely a list of lines and Bézier curves, into some result, such as a path, texture or transformation. To ensure resolution-independence, *analyze* is a constructor rather than a function: in this way we delay the sampling of the path until we know the desired resolution, namely when the renderer runs. We also use this constructor to define resolution independent *constructive solid geometry operations* on shapes, set operations such as union and intersection operating on the set of points inside a shape. The implementation of these operations involves analyzing the intersections between the concrete geometry of both shapes. As an example, the shape in Figure 2(b) can be obtained as follows:

$$\mathit{pacman} = \mathit{subtract}(\mathit{circ}, \mathit{triangle})$$

3.2 Textures

To declare the interior of a shape, a texture can be created with the *texture* constructor, which requires a function from a point to a color. A *color* is a value with four numbers, all in the range $[0, 1]$, namely red, green, blue and alpha (transparency). For example, consider the following colors:

$$\mathit{red} = \mathit{color}(1,0,0,1), \mathit{black} = \mathit{color}(0,0,0,1), \mathit{yellow} = \mathit{color}(1,1,0,1)$$

We can now create a radial gradient as follows:

$$\mathit{radgrad} = \mathit{texture}(\lambda\langle x, y\rangle \rightarrow \mathit{lerp}(\mathit{red}, x^2 + y^2, \mathit{black}))$$

Where *lerp* performs linear interpolation of two colors on each of the four numbers. A *TexturedShape* can then be created using the *fill* constructor. For example, Figure 2(b) shows:

$$\mathit{fill}(\mathit{pacman}, \mathit{radgrad})$$

As another example of defining textures in our framework, consider the interior of the triangle shown in Figure 2(c). For this texture, we first declare a one-dimensional cyclic gradient that cycles between red and yellow:

$$\begin{aligned} \mathit{gradient}(x) = {} & \textbf{if } l \leq \tfrac{1}{2} \textbf{ then } \mathit{lerp}(\mathit{red}, 2 \times l, \mathit{yellow}) \\ & \quad \textbf{else } \mathit{lerp}(\mathit{yellow}, 2 \times (l - \tfrac{1}{2}), \mathit{red}) \\ & \textbf{where } l = x - \lfloor x \rfloor \end{aligned}$$

We can then define the filling of the triangle as follows:

$$\mathit{tritex} = \mathit{texture}(\lambda\langle x, y\rangle \rightarrow \mathit{lerp}(\mathit{gradient}(x \times 10), (2 \times |y|/x)^2, \mathit{black})$$

Where $x \times 10$ repeats the gradient ten times on the horizontal $[0, 1]$ interval and the linear interpolation argument[5] $(2 \times |y|/x)^2$ ensures that the color becomes darker closer to the vertical border of the triangle. A further survey of the power of this way of describing textures is beyond the scope of this paper, for some fascinating examples see [6] and [2].

3.3 Transformations

The *transformation* constructor can be used to describe arbitrary transformations and requires the forward transformation function and its inverse. For example, we can define a scaling transformation as follows:

$$\begin{aligned} \mathit{scale}(s_x, s_y) = \mathit{transformation}(&\lambda\langle x, y\rangle \rightarrow \langle s_x \times x, s_y \times y\rangle, \\ &\lambda\langle x, y\rangle \rightarrow \langle x/s_x, y/s_y\rangle) \end{aligned}$$

We can use this transformation to scale our previous examples. For example, to make our filled triangle half as big, we can do the following:

$$\mathit{transform}(\mathit{scale}(1/2, 1/2), \mathit{fill}(\mathit{triangle}, \mathit{tritex}))$$

Where the *transform* function is expressed as follows:

$$\begin{aligned} &\mathit{transform}(\mathit{transformation}(f, f^{-1}), \mathit{path}(p)) = \mathit{path}(f \circ p) \\ &\mathit{transform}(f, \mathit{shape}(l)) = \mathit{shape}([\mathit{transform}(f, p) \mid p \leftarrow l]) \\ &\mathit{transform}(\mathit{transformation}(f, f^{-1}), \mathit{texture}(t)) = \mathit{texture}(t \circ f^{-1}) \\ &\mathit{transform}(f, \mathit{fill}(s, t)) = \mathit{fill}(\mathit{transform}(f, s), \mathit{transform}(f, t)) \end{aligned}$$

[5] When $x = 0$, $|y|/x$ will be ∞ or not-a-number, which will cause *lerp* to return black.

The only constraint on a transformation is that it must be continuous, otherwise it would be possible to transform a closed path (defining an area) into an open path (not defining an area). As an example of a non-affine transformation consider the "wave" transformation shown in Figure 3(a):

$$\mathit{wave} = \mathit{transform}(\lambda\langle x,y\rangle \rightarrow \langle x+\sin(y),y\rangle), \lambda\langle x,y\rangle \rightarrow \langle x-\sin(y),y\rangle)$$

These transformations can be composed using the following *compose* function, which uses the well-known rule $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$.

$$\mathit{compose}(\mathit{transform}(f,f^{-1}),\mathit{transform}(g,g^{-1})) = \mathit{transform}(f \circ g, g^{-1} \circ f^{-1})$$

A benefit of having both directions of a transformation is that we can also transform *transformations.* For example, if we have a rotation transformation and we want to change the center of rotation, we can achieve this by transforming the rotation by a translation. This is done by first applying the inverse translation, then the rotation and then the forward translation. In general, we can transform any transformation by another transformation as follows:

$$\begin{aligned}&\mathit{transform}(t,r) = \mathit{compose}(t,\mathit{compose}(r,\mathit{inverse}(t)))\\ &\qquad\qquad\textbf{where } \mathit{inverse}(\mathit{transform}(f,f^{-1})) = \mathit{transform}(f^{-1},f)\end{aligned}$$

As an example, we can transform our wave transformation to produce smaller waves:

$$\mathit{scaledWave} = \mathit{transform}(\mathit{scale}(1/30,1/30),\mathit{wave})$$

Applying this transformation to our filled triangle produces Figure 3(a).

Another example of a non-affine transformation is a "sweep": mapping the [0,1] interval on the x-axis to a given path. For example, by first scaling our filled triangle to make it thinner we can obtain Figure 3(b) as follows:

$$\mathit{fspir} = \mathit{transform}(\mathit{compose}(\mathit{sweep}(\mathit{spiral}),\mathit{scale}(1,1/40)),\mathit{ftriangle})$$

Other papers [10,9] have shown how to implement the sweep transformation when only the forward transformation is required, we now show how to handle both directions of this transformation. To define this transformation in a resolution-independent way, we define it as a function which takes the concrete geometry of the path and returns a transformation. Using the *analyze* constructor, we make this function into a transformation.

To prevent changes in speed along the path, we want the norm of the derivative to be constant along the path. To this end, we reparameterize the concrete geometry of the path to a new geometrical description, q, with the same shape and a constant norm of the derivative, using an algorithm such as [11]. The forward transformation can then be expressed as follows:

$$\lambda\langle x,y\rangle \rightarrow q(x) + y \times \widehat{q'(x)}$$

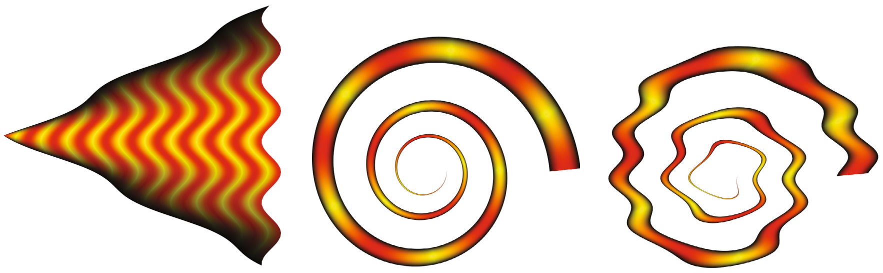

(a) Wave transformed triangle. (b) Triangle swept along spiral. (c) The filled triangle swept by a spiral transformed by a wave.

Fig. 3. Non-affine transformation examples

Here $\hat{x}$ denotes a normalized vector and q' is the derivative of q.

The inverse transformation works by finding the closest point on the path to the point that is to be transformed. The horizontal coordinate is then the parameter at that point on the path, and the vertical coordinate is the distance of the point to be transformed from the path. More precisely:

$$\lambda v \rightarrow \langle t, sgn(q'(t)) \times |q(t) - v| \rangle \ \mathbf{where}\ t = f(v)$$

Here *sgn* is the sign function and f computes the parameter of the closest point on q to a given point, using an algorithm such as [12]. As a final example of the compositionality this framework gives us, we transform the swept triangle using our wave transformation to obtain Figure 3(c):

$$transform(scaledWave, fspir)$$

4 Implementation and Optimization

Our approach can be efficiently implemented by mapping it to a graphics host. We first describe a basic implementation and then introduce some extensions to allow more optimizations. Finally, we show how we can support clipping and discuss potential further optimization. The implementation of Deform as sketched in this section is surprisingly concise and simple and consists of just 983 lines of Scala code.

4.1 Basic Implementation

The main function to implement is the *render* function, which acts as an interpreter for the constructors that may occur in a *TexturedShape*. The pipeline of the *render* function is shown in Figure 4 and is organized as follows; A *TexturedShape* is produced by the user program and its shape is then translated into geometry, i.e., lines and Bézier curves, which are in turn translated to their equivalent representations in the graphics host. The graphics host then fills the shape, producing a raster telling us which pixels are inside the shape. We then simply iterate over these pixels and call the corresponding texture function for each pixel, producing a color raster which is then sent to the display.

The *toBézier* function in this pipeline is also used to interpret *analyze* constructors, namely to generate the concrete geometry which is fed to the function argument of the constructor. We currently use a simple implementation of this function: we sample the function until the samples are so close to each other that the error is smaller that the size of a pixel. Afterwards, the samples are joined by lines.

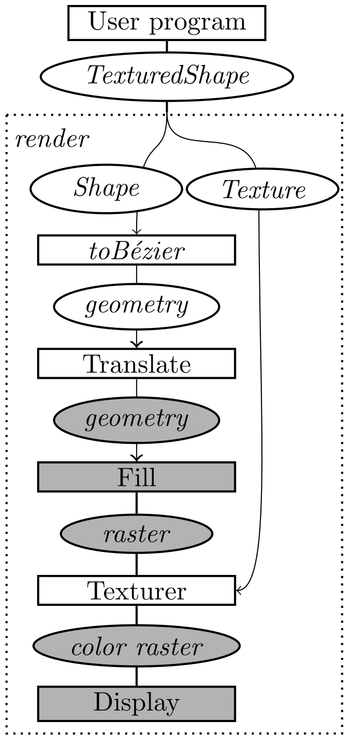


Fig. 4. Rendering pipeline. Gray indicates functionality from the graphics host.

4.2 Special Cases

We optimize the basic implementation by intercepting special cases and mapping them to the corresponding functionality of the graphics host. We add a new constructor for each special case, which are shown in Table 3. Several of these new constructors were presented earlier as functions and by transforming them into constructors the *render* function can recognize them and act accordingly. We now discuss the special cases for shapes, textures and transformations.

Shapes. The first special case for shapes concerns paths that consist of lines and Bézier curves. It is of course wasteful to use a combination of lines and Bézier curves, only to later approximate it with other lines and Bézier curves. Hence, we extend our *Path* type with extra constructors for these types of paths and a constructor for *join*, so that the *toBézier* function can immediately use these descriptions without sampling.

The second special case for shapes deals with constructive solid geometry operations. The default implementation of these operations is to obtain a concrete geometry of the shapes using *toBézier* and then analyze intersections to produce the new shape. In the case of union or symmetric difference we can skip this analysis. The union of a set of shapes can be implemented by supplying the set of shapes to the fill function of the graphics host and using the *non-zero fill rule*. This tells the renderer to fill any pixel that is inside at least one of the shapes, effectively rendering the union of the shapes. Analogously, we can render the symmetric difference of a list of shapes by using the *even-odd fill rule*, which states that a pixel should be filled if it is inside an odd number of shapes.

Textures. If the graphics host has support for a texture, we would like to make use of these optimized capabilities, because then we can completely skip the Texturer step in the pipeline. Hence, we include the constructor *nativeTexture* for these cases, which takes a function that given an affine transformation gives the specific representation for the graphics host of the transformed texture and a regular texture function for use when the transformation of the texture is not affine.

Transformations. If a transformation is affine and the path consists of lines and Bézier curves, we transform the geometry directly, instead of by sampling a function. The constructor *affineTransformation* represents such an affine transformation by two matrices (the specification of this type is left open), one for the forward transformation and one for the inverse transformation. We also change the *transform* function into a constructor so that the *toBezier* function can intercept this special case. The *compose* function is also adapted to intercept the special case of composing an affine transformation with another affine transformation, which can be done using matrix multiplication instead of function composition, saving computations when points are transformed.

Table 3. Additional constructors for special cases

Constructor	*Type*
line	$\mathbb{R}^2 \times \mathbb{R}^2 \to \mathit{Path}$
quadBezier	$\mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \to \mathit{Path}$
cubicBezier	$\mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \to \mathit{Path}$
join	$[\mathit{Path}] \to \mathit{Path}$
union	$[\mathit{Shape}] \to \mathit{Shape}$
symdiff	$[\mathit{Shape}] \to \mathit{Shape}$
nativeTexture	$(\mathit{Matrix} \to \mathit{NativeTextureDesc}) \times (\mathbb{R}^2 \to \mathit{Color}) \to \mathit{Texture}$
transformation	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \to \mathit{Transformation}$
transform	$\mathit{Transformation} \times A \to A$
	where $A \in \{\mathit{Path}, \mathit{Shape}, \mathit{TexturedShape}, \mathit{Transformation}\}$
affineTransformation	$\mathit{Matrix} \times \mathit{Matrix} \to \mathit{Transformation}$
pathbb	$(\mathbb{R} \to \mathbb{R}^2) \times \mathit{BBox} \to \mathit{Path}$
transformationbb	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathit{BBox} \to \mathit{BBox}) \to \mathit{Transformation}$

Performance. Note that in traditional frameworks such as Java2D or Processing, the special cases presented above are the *only* things that are expressible. Thus, the interception of these special cases guarantees that drawings that could also be produced using such a library are approximately as fast. We verified this by generating equivalent Java2D and Deform code in which 100,000 shapes (letters) were rendered, each with their own native texture and affine transformation. The Deform code performed 0.8% slower than the direct Java2D calls. This minor difference in speed is due to the fact that the Deform code first builds an intermediate representation of the textured shapes.

4.3 Clipping

For large scenes, involving many shapes, a valuable optimization is *clipping*: determining the bounding boxes of shapes and then ignoring the shapes that are not in view. However, since in our framework shapes and transformations can be arbitrary functions, it is impossible to discover the bounding box of a shape without sampling it.

For this reason we add two new constructors: one to declare a path and its bounding box (the specification of this type is left open) and one to declare a transformation and also a function to forwardly transform a bounding box. In this way the user can optionally give the bounding boxes of transformed shapes. If the bounding boxes are not supplied, the shapes will simply not profit from clipping. In Deform, all library functions to construct paths and transformations also deal with bounding boxes. For example, lines and Bézier curves get the bounding box induced by their (control) points and *join* produces the smallest bounding box that contains the bounding boxes of its arguments. Affine transformations transform a bounding box by transforming each of its vertices. We currently use axis-aligned bounding boxes, but it is also possible to use non-axis-aligned ones that fit the shapes more tightly, at the cost of more computations.

4.4 Potential Optimization

A potential optimization might be to speed the *toBézier* function by using techniques from the field of curve fitting. We could do the sampling and fitting in parallel, by modifying a curve fitting algorithm such as [13]. We can then stop the sampling earlier if the samples we take lie close enough to the current approximation. We can also use the parameter of each point to improve the speed of our approximation since this is often useful information for curve fitting algorithms [13]. Finally, curve fitting algorithms often estimate a derivative of the shape, so if we numerically compute the derivative, or supply it using an automated differentiation system [14], we can also use this information to more quickly find an approximation of the curve.

5 Case Study: Focus+Context Lenses

As a real world example of how this framework enables advanced, resolution-independent computer graphics techniques in a compositional way, we show how to implement the form of focus+context lenses that are presented in [15], which have been shown to be useful in human computer interaction [16]. A focus+context lens, such as the one in Figure 1, is a transformation that magnifies a part of the space (the focus area) and shows how this magnified part fits into the rest of the space (the context) through a deformation. We compare our implementation to the previous implementation of this form of focus+context lenses [16]. Our implementation is slightly harder, since we require both directions of the transformation. As we will show, this effort is well spent since it yields a faster implementation that gives better image quality at a fraction of the code.

5.1 Implementation

We first consider the *inverse* transformation as presented in [15,16]. Figure 5 shows the elements of a lens: r_f is the radius of the focus area, r_l is the radius of the lens and we define m as the magnification factor. The inverse transformation is then defined as follows:

$$l^{-1}(v) = \begin{cases} v/m & |v| < r_f \\ \frac{v}{|v|} \times n^{-1}(|v|) & r_f < |v| < r_l \\ v & \text{otherwise} \end{cases}$$

Where n^{-1} is the function that describes the deformation, by giving the new norm, i.e., distance from the center of the lens, for the point to be transformed and is a continuous, monotonically increasing function from $[r_f, r_l]$ to $[r_f/m, r_l]$:

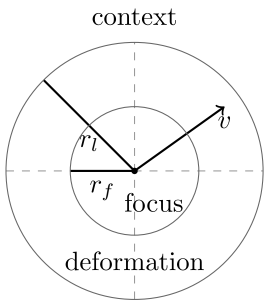


Fig. 5. Lens elements

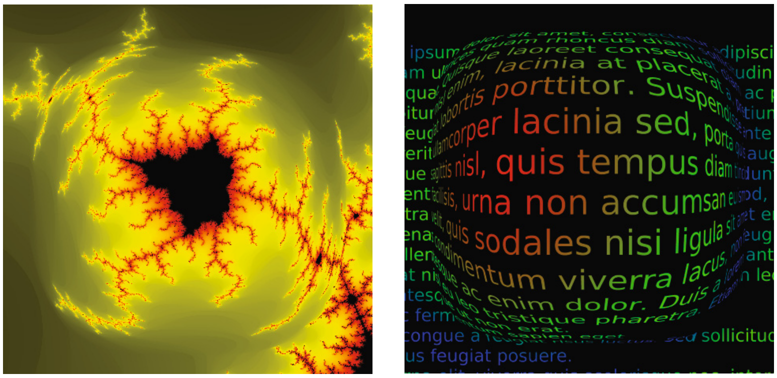
$$n^{-1}(d) = \frac{d}{(1 - p(z)) \times (m-1) + 1} \quad \textbf{where } z = (d - r_f)/(r_l - r_f)$$

Here z describes how far into the deformation area the point is, with zero if the point is on the border of the magnification area and one if it is on the border of the context area. The *profile* function, p, describes the shape of the deformation and can be chosen freely as long as it is a continuous, monotonically increasing function from $[0,1]$ to $[0,1]$, such as the identity function. Another variation point is which norm to use to compute $|v|$, which decides the shape of the lens. In general it is possible to use any L^P norm, which are of the form $\sqrt[p]{x^p + y^p}$. The lens is circular with L^2 and with L^∞ the norm resolves to $\max(x,y)$ and the lens is square. The example in Figure 1 uses the Euclidian norm and a Gaussian profile function and Figure 6 shows two more Deform screenshots of other lenses in action.

We now need to derive the *forward* transformation from this inverse transformation. If we have the inverse of the function n^{-1}, then the forward transformation can be expressed as follows:

$$l(v) = \begin{cases} v \times m & |v| < r_i/m \\ \frac{v}{|v|} \times n(|v|) & r_f/m < |v| < r_l \\ v & \text{otherwise} \end{cases}$$

However, for many profile functions, there is no analytic solution for the inverse of n^{-1}. Luckily, n^{-1} is a continuous monotonically increasing function, so we can implement $n(t)$ by numerically searching for the x such that $n^{-1}(x) = t$. We use Newton's method for this, since it is very efficient at finding the roots of monotonic functions. This method requires the derivative of n^{-1}, which can be constructed using the derivative of the profile. In this way only the profile function and its derivative are needed when creating a lens with a different profile.



(a) L^3 norm, linear profile (b) L^4 norm, quadratic profile

Fig. 6. Different types of lenses in action

5.2 Comparison

The previous implementation [16] of this form of focus+context lenses is in the Zoomable Visual Transformation Machine (ZVTM) [17] framework for zoomable user interfaces. The advantage of their approach to implementing these lenses is that it is very loosely coupled with the graphics host, and is thus applicable in many graphical frameworks. In our approach these lenses can be added easily and this yields a better implementation in terms of length of code, speed and image quality.

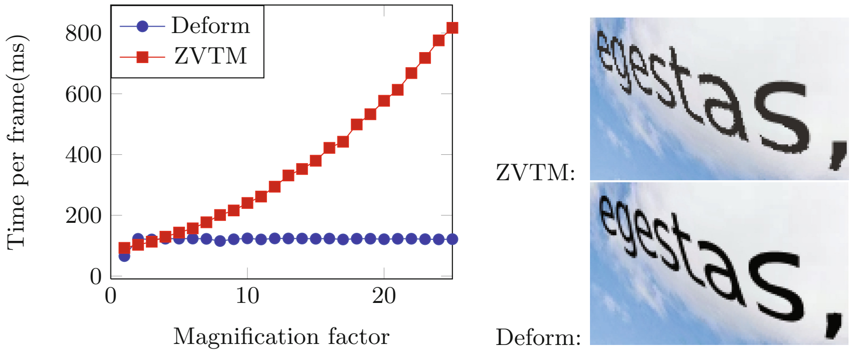
Code Size. In the ZVTM implementation, defining the lenses requires about 700 lines of code, and each new lens (with a different norm or profile) requires about 100 lines of code [16]. In our declarative framework, the implementation of these lenses requires 43 lines of code, including the definition of the (reusable) numeric approximation code, while defining a new lens can be done in a single line of code. For example, a rounded square lens with a quadratic profile (with derivative $2\times x$), as shown in Figure 6(b), is declared as follows:

$$\mathit{lens}(\lambda\langle x, y\rangle \to \sqrt[4]{x^4 + y^4}, \lambda x \to x^2, \lambda x \to 2\times x)$$

Performance. As a performance comparison, we implemented the setup shown in Figure 1 in both Deform and ZVTM and measured the time it took to render a single image at different magnification factors. This was chosen because it is a simple example of a combination of shapes (text) and a texture (bitmap image). The entire picture was 1600x1000 pixels big and the lens had a focus radius of 100 pixels and a lens radius of 200 pixels. Note that both ZVTM and Deform run on the JVM and are built on top of Java2D. Figure 7(a) shows the results of our measurements on an Intel i7 2.8GHz CPU running OpenJDK 1.11.3. All measurements are the average of 100 runs.

We can see that in ZVTM the magnification factor has a huge impact on performance, whereas in Deform it has no effect at all. This is because ZVTM does not feature non-affine transformations in general and uses a trick to achieve focus+context lenses; It renders the lens area *twice*: once without magnification and once with magnification. Afterwards, both renderings are sampled to produce the lens area. The second, magnified rendering uses a buffer of width and height $2\times m\times r\_l$. Hence the amount of pixels in this buffer is $(2\times m\times r\_l)^2$, which explains the quadratic growth of the ZVTM rendering time.

Image Quality. As a final comparison, we consider the image quality of both approaches as shown in Figure 7(b). This notable difference in image quality is caused by the fact that Deform performs the discretization of shapes and textures later. ZVTM performs the discretization *before* applying the lens, while



(a) Difference in speed. (b) Difference in image quality.

Fig. 7. Performance and image quality comparison

Deform performs the discretization *after* applying the lens. Hence Deform does not suffer from aliasing artifacts.

6 Discussion

While our framework is very expressive, it currently does not support post-processing image filters such as blurs. These filters are computationally very expensive and require low-level optimizations for real-time performance. Halide [18] is an example of a language that is specifically designed for such filters; the programmer gives a concise declarative description of the filter along with a schedule that states how the filter must be implemented. This yields very good results, outperforming hand tuned assembly code in some cases. It would be interesting to explore how the Halide way of describing filters can be fitted into our framework.

Another open question is how we can exploit the massive power that is available via GPUs: which paths, transformations and textures can be executed on the GPU and how? How can these parts work together with functionality that cannot be executed on the GPU? Answering these questions will lead to a truly high-performance implementation of Deform.

7 Conclusion

We have presented a novel declarative framework for resolution-independent 2D graphics that is simple, expressive and composable while still being applicable to real-life graphics. We have shown how to implement this framework such that it easily maps to readily available, highly-optimized procedural graphics libraries and have also shown how this framework can support clipping, so that it is possible to render very large scenes. We have shown a simple benchmark that shows that our framework is as fast as directly using the graphics host, thanks to the interception of special cases. As a real-world example, we have implemented focus+context lenses. The result is faster and smaller than the state-of-the-art implementation and has better image quality. Our framework liberates the programmer from the limitations of traditional frameworks and we expect that it forms an excellent foundation for creating resolution-independent graphics and higher-level visualization tools in a wide range of domains.

Acknowledgements. We thank Robert van Liere and Tijs van der Storm for their helpful comments.

References

1. Klint, P., Lisser, B., van der Ploeg, A.: Towards a one-stop-shop for analysis, transformation and visualization of software. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 1–18. Springer, Heidelberg (2012)

2. Karczmarczuk, J.: Functional approach to texture generation. In: Adsul, B., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 225–242. Springer, Heidelberg (2002)
3. Hoffmann, C.M.: Implicit curves and surfaces in CAGD. IEEE Computer Graphics and Applications 13, 79–88 (1993)
4. Sederberg, T.W., Anderson, D.C., Goldman, R.N.: Implicit representation of parametric curves and surfaces. Computer Vision, Graphics, and Image Processing 28(1), 72–84 (1984)
5. Dokken, T., Thomassen, J.: Overview of approximate implicitization. In: Topics in Algebraic Geometry and Geometric modelling. AMS series on Contemporary Mathematics CONM 334, vol. 28(1), pp. 169–184 (2003)
6. Elliott, C.: Functional image synthesis. In: Proceedings of Bridges (2001)
7. Finne, S., Jones, S.P.: Pictures: A simple structured graphics model. In: Glasgow Functional Programming Workshop, Ullapool (1995)
8. Matlage, K., Gill, A.: ChalkBoard: Mapping functions to polygons. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 55–71. Springer, Heidelberg (2010)
9. Elliott, C.: Programming graphics processors functionally. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell 2004, pp. 45–56. ACM, New York (2004)
10. Karczmarczuk, J.: Geometric modelling in functional style. In: Proceedings of the III Latino-American Workshop on Functional Programming, CLAPF 1999, pp. 8–9 (1999)
11. Casciola, G., Morigi, S.: Reparametrization of NURBS curves. International Journal of Shape Modeling 2, 103–116 (1996)
12. Ma, Y.L., Hewitt, W.T.: Point inversion and projection for NURBS curves and surfaces: control polygon approach. Comput. Aided Geom. Des. 20(2), 79–99 (2003)
13. Schneider, P.J.: An algorithm for automatically fitting digitized curves. In: Glassner, A.S. (ed.) Graphics gems, pp. 612–626. Academic Press Professional, Inc., San Diego (1990)
14. Elliott, C.: Beautiful differentiation. In: International Conference on Functional Programming, ICFP (2009)
15. Carpendale, M.S.T., Montagnese, C.: A framework for unifying presentation space. In: Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology, UIST 2001, pp. 61–70. ACM, New York (2001)
16. Pietriga, E., Bau, O., Appert, C.: Representation-independent in-place magnification with Sigma lenses. IEEE Transactions on Visualization and Computer Graphics 16, 455–467 (2010)
17. Pietriga, E.: A Toolkit for Addressing HCI Issues in Visual Language Environments. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 145–152 (2005)
18. Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., Durand, F.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Transactions on Graphics 31(4), 32 (2012)

Analysing the Entire Wikipedia History with Database Supported Haskell

George Giorgidze[1], Torsten Grust[1], Iassen Halatchliyski[2], and Michael Kummer[3]

[1] University of Tübingen, Germany
[2] Knowledge Media Research Center, Tübingen, Germany
[3] Centre for European Economic Research, Mannheim, Germany

Abstract. In this paper we report on our experience of using Database Supported Haskell (DSH) for analysing the entire Wikipedia history. DSH is a novel high-level database query facility allowing for the formulation and efficient execution of queries on nested and ordered collections of data. DSH grew out of a research project on the integration of database querying capabilities into high-level, general-purpose programming languages. It is an emerging trend that querying facilities embedded in general-purpose programming languages are gradually replacing lower-level database languages such as SQL as preferred facilities for querying large-scale database-resident data. We relate this new approach to the current practice which integrates database queries into analysts' workflows in a rather ad hoc fashion. This paper would interest early technology adopters interested in new database query languages and practitioners working on large-scale data analysis.

1 Introduction

Relational database systems provide scalable and efficient query processing capabilities for complex data analysis tasks. Despite these capabilities, database systems often do little more than to hold and reproduce data items for further processing and analysis with the help of a programming language. In this typical setup, the lion share of data analysis tasks thus happens outside the realm of the database system. This is partly because, for involved analyses, relational database systems require the mastering of advanced features of specialised database query languages, such as SQL. This requirement represents a barrier to many practitioners who need to analyse large-scale data. In this paper, we report on what this means for social scientists interested in Wikipedia data.

Unfortunately, transferring database-resident data into the runtime heap of a programming language is not always an option (for example, if the data size is larger than the available main memory). Even if the transfer is possible, it may be wasteful if the final result of the computation is small. One approach addressing the aforementioned problems is to use relational database systems and their query processing capabilities as *coprocessors for programming languages* [4]. This approach entails the automatic translation of the data-intensive parts of a given program into relational queries (SQL statements, say) that can be executed by a database engine. In effect, the relational database system transparently participates in program evaluation. Most importantly, data-intensive computations are performed by the database engine and thus close to the data.

Under this approach, SQL is regarded as a lower-level target language into which more familiar and expressive source languages are translated. It is an emerging trend to rely on higher-level querying facilities for the analysis of large-scale database-resident data. Perhaps the most widely used example of such a query facility is LINQ [10] which seamlessly adds advanced query constructs to the programming languages of Microsoft's .NET framework.

In this paper we report on our experience of using *Database Supported Haskell* (DSH) [4] for analysing the entire history of Wikipedia. DSH is a novel high-level database query facility allowing for the formulation and efficient execution of queries over database-resident collections of data. Support for nested and ordered data structures as well as powerful programming abstractions is what distinguishes DSH from other language-integrated database query facilities.

DSH grew out of a research project on the tight integration of database querying capabilities into high-level, general-purpose programming languages. While the project tackles foundational issues of the embedding and compilation of rich query languages [7,4,5], here we report on our efforts to team up with colleagues of the social sciences to address topics in large-scale Wikipedia analysis. The use cases of the following sections are taken from current studies on the collaborative construction of knowledge [8,9].

This paper would interest early technology adopters interested in new database query languages and practitioners working on large-scale data analysis. The remainder of the paper is structured as follows. In Section 2, we describe an earlier study performed by the third author on a small subset of the Wikipedia data. The data analysis tasks of this study were implemented in terms of a (what we assume to be typical) ad hoc embedding of SQL queries into R scripts. In Section 3, we describe how we scaled the study to the entire Wikipedia data using DSH. In Section 4, we outline future work and conclude the paper.

2 A Bilingual Approach Based on SQL and R

In a study by the third author of the present paper the development of new knowledge in Wikipedia was analysed at the level of knowledge domains [8]. The study was based on 4,733 articles and 4,679 authors in the equally large adjacent knowledge domains of physiology and pharmacology. The *boundary spanners* were shown to be a highly relevant group of authors for the whole knowledge-creating community: these authors work on integrative boundary articles between domains and also on the central articles within a single domain.

The analysis has been implemented in terms of a mixture of two main technologies, SQL and R. Relevant data was sourced from a MySQL database dump of the German Wikipedia that contained all logged data on articles and authors. Multiple SQL queries were textually embedded into a single script of R code

```
1  library(RODBC)
2  con <- odbcConnect("...")
3  # retrieve articles in the physiology and pharmacology domains
4  phyA <- dbGetQuery(con,"SELECT pd_page_id FROM pagedomains
5                              WHERE (pd_domain = 'physiology')")
6  phaA <- dbGetQuery(con,"SELECT pd_page_id FROM pagedomains
7                              WHERE (pd_domain = 'pharmacology')")
8  allA <- unique(rbind(phyA,phaA))
9  # Establish direct links between the retrieved articles
10 links <- dbGetQuery(con,
11    paste( "SELECT pl_from, page_id FROM pagelinks, page
12                WHERE pl_title = page_title
13                  AND page_namespace = 0
14                  AND page_is_redirect = 0
15                  AND pl_from IN (", toString(allA[,1]), ")
16                  AND page_id IN (", toString(allA[,1]), ")", sep = ""))
```

Fig. 1. R script snippet supporting the analysis of boundary-spanning articles and authors in the German Wikipedia

that drove the analysis. We consider this to be a representative setup. Similar studies have used other host languages (e.g., Perl, Python, and C). Our general observations below remain valid, however. It is inherent to this approach that the analyst is *bilingual*, capable of speaking two languages and also fit to translate logic as well as data formats between the two (SQL and host language).

Figure 1 displays a snippet from the mentioned R script. The code is sprinkled with SQL fragments: the variable assignment `phyA <- dbGetQuery(con,...)` sends the quoted SQL statement to a connected database server for execution and binds the resulting n-ary table to the R variable `phyA`, a data frame (or matrix) of n columns.

A number of issues make this style of data-intensive programming problematic for the analyst. Among the more pressing issues are the following:

(Lack of) Static Safety. R does not understand the quoted SQL text and sends it as is. The text may contain syntactically invalid or even harmful SQL code [13]. This is important if fragments of SQL text are spliced at script runtime. The lack of static safety and the possibility of run time failures is particularly problematic for long running programs as a runtime error may require a restart of the entire computation, including those (potentially long-running) parts executed before the failure occurred.

Query Text Size and Number of Queries. Note how the `toString()` calls in lines 15 and 16 refer to a formerly computed query result (R variable `allA`). The size of these splices, and thus of the overall generated query text, depends on the queried data and thus must, in principle, be considered arbitrarily large. In effect, intermediate query results (here bound to `allA`) are carried from query to query in textual form. Besides the obvious inefficiency, this approach may easily overwhelm the database system's SQL parser or compiler: both `IN` clauses

Table 1. Wall-clock execution times for the R and DSH program fragments from Sections 2 and 3

# Articles	**SQL+R**	**DSH**
	⏱ (sec)	⏱ (sec)
10,000	46.25	1.15
20,000	108.89	2.97
30,000	181.63	3.80
40,000	263.77	4.64
50,000	353.31	5.54

in line 15 and 16 end up containing almost five thousand literals (the aforementioned 4,733 page identifiers, to be more precise).

Alternatively, to avoid the construction of huge queries, carried intermediate results may be used to issue multiple separate but simple queries in an iterative fashion. This mode of data-intensive computation, however, incurs considerable run time overhead associated with the now frequent switches between R's runtime system and the database system [7,4]. Note that in this latter scenario, the boundary spanner analysis would lead to at least 4,733 of these costly context switches. Scalability clearly suffers.

Computation Outside the Database Realm. The various SQL fragments contribute sizeable intermediate results which are materialised on the R heap and then threaded through the script. R operations (e.g., `unique`: duplicate elimination, `rbind`: union, `[,]`: projection) are used to perform data-intensive computation on the R heap while the database server would have been perfectly able to do the job more efficiently close to the source data. Again, this raises severe scalability and performance issues.

Host Language Dictates Execution Granularity and Order. Following a typical style, the R script breaks the data-intensive portion of the computation down into parts that lead to intelligible pieces of SQL code. The synchronous execution of these pieces through `dbGetQuery()` prescribes a granularity and order of query evaluation that leaves little room for query optimisation and prevents query scheduling by the database engine.

The small snippet of R code in Figure 1 exhibits all four issues mentioned above. These issues are instances of problems of programming language and database interoperability that have long been identified [3]. The present paper was motivated by the assumption that vibrant areas of data-intensive research, such as the Wikipedia analysis, would particularly benefit from a modern account of database integration.

3 A Unilingual Approach Based on DSH

Figure 2 gives DSH definitions corresponding to the R script from Figure 1. DSH allows for formulation of database executable program fragments using Haskell's

```
phyA :: Q [Integer]
phyA = [pd_page_idQ p | p ← pageDomains, pd_domainQ p ≡ "physiology"]
phyA :: Q [Integer]
phyA = [pd_page_idQ p | p ← pageDomains, pd_domainQ p ≡ "pharmacology"]
allA :: Q [Integer]
allA = nub (phyA ⧺ phaA)
links :: Q [(Integer, Integer)]
links = [ tuple (pl_fromQ l, page_idQ p)
        | p ← pages
        , l  ← pageLinks
        , pl_titleQ l ≡ page_titleQ p
        , page_namespaceQ p ≡ 0
        , page_is_redirectQ  p ≡ 0
        , pl_fromQ l ∈ allA
        , page_idQ p ∈ allA]
```

Fig. 2. DSH definitions corresponding to the R script given in Figure 1

list prelude and the monad comprehension notation [5]. The former turns DSH into an expressive database query facility for nested and ordered collections of data, while the later ensures that DSH queries can be expressed concisely in a widely understood and adopted notation.

Unlike the R script, *all* DSH definitions are database-executable. This provides significant performance benefits as shown in Table 1. In this benchmark, the increase in the number of Wikipedia articles imitates the scenario where the total number of articles in the two domains of interest is much larger than in physiology and pharmacology. As for the rest of the evaluation setup, we used: the latest complete German Wikipedia database dump (June 3, 2012) with 3,958,157 entries in the `page` table and 85,969,266 entries in the `pagelinks` table, version 9.1.4 of the PostgreSQL database management system, the Arch Linux distribution with kernel-3.4.4, version 2.15.0 of the R system, version 7.4.1 of the Glasgow Haskell Compiler (GHC), and a host equipped with an Intel Xeon X5570 CPU.

How does DSH-based data analysis fare if compared with the widely deployed ad hoc embedding approach? Have the major issues of bilingual data analysis actually been addressed?

Static Safety. DSH's query compiler guarantees that the translation of database-executable program fragments will not fail at run time and the translation will generate valid database-executable SQL queries. If a computation has been tagged with the type constructor Q but cannot actually be performed inside the database system DSH will reject the program right from the start. No analyst time is wasted due to late failure at analysis run time.

Query Text Size and Number of Queries. With DSH, the number of generated database queries and their query text size does *not* depend on the size of the queried data. In fact, the number of generated SQL queries is predictable: it is statically determined by the *type* of the database-executable fragment. Specifically, the number of generated queries equals the number of list type constructors (i.e., $[\cdot]$) in the type. For details about the compilation technology that provides this essential guarantee the reader may refer to [7,4].

Computation Outside the Database Realm. DSH allowed us to perform all of our analysis using the underlying relational database system, close to the data. This was instrumental to scale our analysis to the entire Wikipedia history in our follow-up study investigating an economic view of knowledge creation [9].

Host Language Dictates Execution Granularity and Order. DSH provides the guarantee that values of Q-types are evaluated by the database coprocessor without unnecessary context switches between the host language runtime system and the relational database management system. This allows the database to determine the best possible query evaluation strategy as it has access to the entire query and not just parts of the query.

4 Further Reading, Conclusions and Future Work

Due to the limited space we are not able to discuss all of DSH's distinguished features in this paper; particularly its ability to handle nested and ordered collections, and its embedding and compilation aspects. For these topics the interested reader may refer to [7,4,5]. TryDSH, which is an interactive web-based environment for writing, executing, and inspecting the compilation pipeline of DSH queries, can be accessed from [2]. The source code of DSH is available from [1].

In this paper we reported on how we use DSH for large-scale data analysis of Wikipedia data. We observed that the widely used combination of SQL queries interleaved with data analysis in programming languages such as R does not scale for large data sets such as the entire revision, interlinkage, and page access history of the German Wikipedia. DSH allowed us to analyse this large data set from a high level of abstraction and, at the same time, perform the analysis almost entirely using a relational database management system.

One problem that we encountered while using DSH for the Wikipedia data analysis is that, in some DSH queries, we found it hard to reason about the performance behaviour of the resulting SQL queries. This is partly because of the current somewhat involved automatic translation of high-level DSH constructs to lower-level relational database languages. Although we managed to overcome the aforementioned problems with careful reformulations of the DSH queries, a more systematic solution involving changes in the DSH query compiler and optimiser would be beneficial. This would be yet another step towards our goal of allowing practitioners—who are not necessarily expert programmers or database engineers—to analyse large-scale data, such as the entire Wikipedia history, using high-level and reusable domain-specific languages and libraries.

This paper focuses on DSH, which allows a relational database management system to be used as a coprocessor for the Haskell programming language. It is worthwhile to mention that the query compilation techniques featured in DSH are also being used to improve database query facilities in other languages such as C# [7], Ruby [6], Links [11] and Scala [12]. We conjecture that the tight integration of general-purpose programming languages with relational database management systems is a trend that will continue.

Acknowledgements. The authors work within an interdisciplinary cluster of researchers developing an infrastructure for the analysis of social networks. This work is funded by a grant from the ScienceCampus Tübingen. We would like to thank Manfred Knobloch, Thorsten Doherr and Frederic Schütz for helping us with various aspects of sourcing and processing a part of the data used for the analysis.

References

1. Database Supported Haskell (DSH), `http://hackage.haskell.org/package/DSH`
2. TryDSH, `http://dbwiscam.informatik.uni-tuebingen.de/trydsh/`
3. Copeland, G., Maier, D.: Making Smalltalk a database system. ACM SIGMOD Record 14(2), 316–325 (1984)
4. Giorgidze, G., Grust, T., Schreiber, T., Weijers, J.: Haskell boards the Ferry: Database-supported program execution for Haskell. In: Hage, J., Morazán, M.T. (eds.) IFL. LNCS, vol. 6647, pp. 1–18. Springer, Heidelberg (2011)
5. Giorgidze, G., Grust, T., Schweinsberg, N., Weijers, J.: Bringing back monad comprehensions. In: Proc. of the Haskell Symposium 2011. ACM, Tokyo (2011)
6. Grust, T., Mayr, M.: A deep embedding of queries into Ruby. In: Proc. of ICDE 2012, IEEE, Washington, DC (2012)
7. Grust, T., Rittinger, J., Schreiber, T.: Avalanche-safe LINQ compilation. Proc. VLDB Endow. 3(1-2), 162–172 (2010)
8. Halatchliyski, I., Moskaliuk, J., Kimmerle, J., Cress, U.: Who integrates the networks of knowledge in Wikipedia? In: Proc. of WikiSym 2010. ACM (2010)
9. Kummer, M., Saam, M., Halatchliyski, I., Giorgidze, G.: Centrality and content creation in networks the case of German Wikipedia. Technical Report 12-053, ZEW, Mannheim, Germany (2012)
10. Meijer, E., Beckman, B., Bierman, G.: LINQ: reconciling object, relations and XML in the.NET framework. In: Proc. of SIGMOD 2006. ACM (2006)
11. Ulrich, A.: A Ferry-based query backend for the Links programming language. Master's thesis, University of Tübingen (2011)
12. Vogt, J.: Type safe integration of query languages into Scala. Master's thesis, RWTH Aachen University (2011)
13. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proc. of PLDI 2007. ACM, San Diego (2007)

LogicObjects: Enabling Logic Programming in Java through Linguistic Symbiosis[⋆]

Sergio Castro[1], Kim Mens[1], and Paulo Moura[2]

[1] ICTEAM Institute, Université catholique de Louvain, Belgium
{sergio.castro,kim.mens}@uclouvain.be
[2] Center for Research in Advanced Computing Systems, INESC–TEC, Portugal
pmoura@inescporto.pt

Abstract. While object-oriented programming languages are good at modelling real-world concepts and benefit from rich libraries and developer tools, logic programming languages are well suited for declaratively solving computational problems that require knowledge reasoning. Non-trivial declarative applications could take advantage of the modelling features of object-oriented programming and of the rich software ecosystems surrounding them. Linguistic symbiosis is a common approach to enable complementary use of languages of different paradigms. However, the problem of concepts *leaking* from one paradigm to another often hinders the applicability of such approaches. This issue has mainly been reported for object-oriented languages participating in a symbiotic relation with a logic language. To address this issue, we present *LogicObjects*, a linguistic symbiosis framework for transparently and (semi-) automatically enabling logic programming in Java, that aims to solve most of the problems of *paradigm leaking* reported in other works.

Keywords: Linguistic Symbiosis, Object-Oriented Programming, Logic Programming, Multi-Paradigm Programming.

1 Introduction

Object-oriented languages like Java have demonstrated their usefulness for modelling real-world concepts. In addition, the availability of continuously growing software ecosystems around them, including advanced IDEs and extensive libraries, has contributed to their success. Declarative languages like Prolog are more convenient for expressing problems of declarative nature, such as expert systems [1,2]. Linguistic symbiosis [3] has been used in the past to solve the problem of integrating programs written in different languages [1]. Some limitations and issues when implementing such symbiosis, mainly from the point of view of the object-oriented language, have been highlighted in [4] and referred to as *paradigm leaking*. Building upon an earlier position paper [5], this work

⋆ Work partially supported by the LEAP project (PTDC/EIA-CCO/112158/ 2009), the ERDF/COMPETE Program and by the FCT project FCOMP-01-0124-FEDER-022701.

presents a framework that overcomes most of these limitations, while providing a transparent, (semi-)automatic and customisable integration from the perspective of the object-oriented language. New in this paper are the introduction of improved mechanisms for automatic adaptation of logic routine results in the object-oriented world, a context dependent mapping of Java objects to multiple representations in Prolog, and a general mechanism for expressing Java objects in a convenient logic representation even in pure object-oriented programs. We validate our technique by comparing it to the well-known JPL library [6] for invoking logic routines from Java and illustrate the reduction in programming effort.

This paper is structured as follows. Section 2 presents our running example and a logic programming solution. Sections 3 and 4 present our framework and how it enables a transparent and automated access from Java to our implementation in logic. Section 5 discusses related work and Section 6 concludes and presents future work.

2 Case Study: *The London Underground*

Our running example addresses a typical problem that can be implemented easily with a logic language: a querying system about subway lines and stations. But public transportation systems also require a user-friendly interface, which can be developed more easily in an object-oriented language. Therefore, this is a typical case where we can profit from a symbiotic integration between Prolog and Java. In this section, we present a straightforward implementation of our example application in a logic language, discuss how common approaches typically would integrate its logic routines in an object-oriented language, and give an intuitive introduction to our approach and its advantages over current techniques.

Implementation in Logic. The first stage of the problem consists in expressing our knowledge about the London Underground as a set of logic statements. Most of the code in this section has been adapted from [7], with an interesting variation: instead of implementing it in plain Prolog, we use Logtalk [8], a portable object-oriented layer on top of Prolog, facilitating in this way the mapping that needs to be made between objects belonging to each of the two worlds.

```
:- object(metro).
  :- public([connected/3, nearby/2, reachable/3, line/1]).

  connected(station(green_park), station(charing_cross), line(jubilee)).
  connected(station(bond_street), station(green_park), line(jubilee)).
  connected(station(bond_street), station(oxford_circus), line(central)).
  ...

  nearby(S1, S2) :- connected(S1, S2, _).
  nearby(S1, S2) :- connected(S1, S3, L), connected(S3, S2, L).

  reachable(S1, S2, []) :- connected(S1, S2, _).
  reachable(S1, S2, [S3| Ss]) :- connected(S1, S3, L), reachable(S3, S2, Ss).

  line(Name) :- setof(L, S1^S2^connected(S1,S2,L), Ls), list::member(line(Name),
       Ls).
:- end_object.
```

Listing 2.1. The `metro` object in Logtalk

In our example, stations are *connected* to other stations by underground lines. A station is *nearby* another one if there is at most one station in between them. A station A is *reachable* from another station B if there exists a list of stations L that form a path going from B to A. Listing 2.1 shows the Logtalk definition of the `metro` object.[1] The `metro` object encapsulates the knowledge about how stations are connected, plus the rules for the logic predicates `nearby/2`, `reachable/3` and `line/1`. The messages (queries) that the `metro` object can respond to are specified by the `public/1` directive in line 2. Messages in Logtalk are sent using the `::/2` operator, as illustrated on line 15 for the `member/2` method.

```
1 :- object(line(_Name)).
2   :- public([connects/2]).
3
4   connects(S1, S2) :- self(Self), metro::connected(S1, S2, Self).
5 :- end_object.
```

Listing 2.2. The `line` object in Logtalk

Listing 2.2 shows the definition of a *parametric object* [9], `line/1`, which encapsulates the operations of an object representing an underground line. The object parameter denotes the name of the *line*. A `connects/2` predicate (line 4) answers stations directly connected by the *line* object receiving the message. The method implementation is delegated to the `metro` prototype object.

```
:- object(station(_Name)).
  :- public([connected/2, nearby/1, reachable/2]).

  connected(S, L) :- self(Self), metro::connected(Self, S, L).

  nearby(S) :- self(Self), metro::nearby(Self, S).

  reachable(S, IStations) :- self(Self), metro::reachable(Self, S, IStations).
:- end_object.
```

Listing 2.3. The `station` object in Logtalk

Our last object is the `station` object (Listing 2.3). As for the `line` object it is also a parametric object having as sole parameter the name of a station. It defines a method `connected/2` that unifies its first parameter with a station that is connected to this `station` object, through the underground line unified with the second parameter. The method `nearby/1` answers if this station is nearby another station received as a parameter. The method `reachable/2` unifies its first parameter with a station that is reachable from this station object, through a list of intermediate stations unified with the second parameter. As with the `line` object, methods in this `station` object delegate to the `metro` object.

Integration of Logic Routines in an Object-Oriented Language. Most approaches for integrating logic routines in an object-oriented language rely on an explicit mapping between the artefacts of the two worlds. Notions such as a *logic engine*, *logic terms*, *queries*, and *query results* are explicitly represented in

[1] Note that we are defining a *prototype* instead of a *class* as we would do in Java. Although Logtalk also supports classes, using a prototype is simpler in this case.

the object-oriented programs. In the best case, the mappings of these artefacts are simple to implement, but tend to clutter the object-oriented applications that use them with significant boilerplate code that is not related to the core functionality of the application, obscuring in this way its understanding and further evolution. As a representative example of such approaches, we show how a logic routine can be invoked from within Java using the JPL library.

Listing 2.4 shows a partial implementation of a Java class `Station` that uses this library. We include the `connected(Line)` method (lines 14–27) together with required mapping methods. This Java method delegates to the `connected/2` method of the `station/1` Logtalk object (Listing 2.3, line 4). For brevity, we do not discuss here all the details of this JPL-based implementation but we highlight that it contains no less than 14 lines of code just for dealing with mapping tasks. Furthermore, these mapping tasks rely on the existence of auxiliary adapter methods like `asTerm()` and `create(Term)` that are required everywhere we need to adapt a Java object to the logic world and back. In more complex examples, the required boilerplate code can be even more significant.

```
 1 public class Station {
 2     String name;
 3     ...
 4     //mapping an instance of Station to a logic Term
 5     public Term asTerm() {
 6         return new Compound("station", new Term[] {new Atom(name)});
 7     }
 8     //mapping a logic Term to an instance of Station
 9     public static Station create(Term stationTerm) {
10         String lineName = ((Compound)stationTerm).arg(1).name();
11         return new Station(lineName);
12     }
13     //mapping a Java method to a Logtalk method
14     public Station connected(Line line) {
15         Station connectedStation = null;
16         String stationVarName = "Station";
17         Term[] arguments = new Term[]{new Variable(stationVarName),
               line.asTerm()};
18         Term message = new Compound("connected", arguments);
19         Term objectMessage = new Compound("::", new Term[] {asTerm(), message});
20         Query query = new Query(objectMessage);
21         Hashtable<String, Term> solution = query.oneSolution();
22         if(solution != null) {
23             Term connectedStationTerm = solution.get(stationVarName);
24             connectedStation = create(connectedStationTerm);
25         }
26         return connectedStation;
27     }
28     ...//other methods mapped to logic routines
29 }
```

Listing 2.4. The `Station` class in Java using JPL

Towards a Conceptual Mapping with LogicObjects. Our framework provides an alternative to avoid such explicit boilerplate mapping code. As a first example, lines 7–8 of Listing 2.5 show how the `connected(Line)` Java method gets reduced to two lines of code: the method declaration and one annotation.

```
1 @LObject(args = {"name"})
2 public abstract class Station {
3   String name;
4   public Station(String name) { this.name = name; }
5
6   //answers a station connected to this station by means of a line
7   @LMethod(args = {"LSolution", "$1"})
8   public abstract Station connected(Line line);
9
```

```java
10   //answers the list of nearby stations
11   @LComposition @LMethod(args = {"LSolution"})
12   public abstract List<Station> nearby();
13
14   //answers the list of intermediate stations between this and another station
15   @LMethod(name = "reachable", args = {"$1", "LSolution"})
16   public abstract List<Station> intermediateStations(Station station);
17 }
```

Listing 2.5. The `Station` class in Java using LogicObjects

The `Station` class is the Java counterpart of the `station/1` Logtalk object defined in Listing 2.3. It declares a `name` member variable (line 3) denoting the name of the underground station. The `Line` class (Listing 2.6) is the Java counterpart of the `line/1` Logtalk object defined in Listing 2.2. It declares a `name` member variable denoting the name of the underground line.

```java
@LObject(args = {"name"})
public abstract class Line {
  String name;
  public Line(String name) { this.name = name; }

  //answers if two stations are connected by this line
  public abstract boolean connects(Station s1, Station s2);

  //answers the number of stations connected by this line
  @LMethod(name = "connects", args = {"_", "_"})
  public abstract int segments();
}
```

Listing 2.6. The `Line` class in Java using LogicObjects

Finally, the `Metro` class (Listing 2.7) is the Java counterpart of the `metro` Logtalk object defined in Listing 2.1.

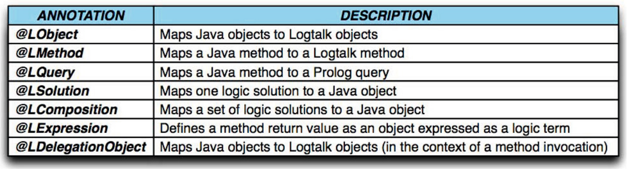
```java
public abstract class Metro {
  //answers a list with all lines
  @LComposition @LMethod(name="line", args={"L"})
  public abstract List<Line> lines();

  //answers an existing line with a given name
  public abstract Line line(String s);
}
```

Listing 2.7. The `Metro` class in Java using LogicObjects

3 LogicObjects

In this section we describe the linguistic symbiosis techniques employed by our *LogicObjects* framework. Figure 1 lists all the annotations currently supported. Our current implementation focusses on a symbiosis from the Java point of view. We decided to design and implement our symbiosis from the perspective of the object-oriented language, since this is the direction that has been reported [1,4] as the most difficult to achieve transparently and automatically. We start our discussion by describing the linguistic symbiosis problems we are going to solve in the remainder of this section.



ANNOTATION	DESCRIPTION
@LObject	Maps Java objects to Logtalk objects
@LMethod	Maps a Java method to a Logtalk method
@LQuery	Maps a Java method to a Prolog query
@LSolution	Maps one logic solution to a Java object
@LComposition	Maps a set of logic solutions to a Java object
@LExpression	Defines a method return value as an object expressed as a logic term
@LDelegationObject	Maps Java objects to Logtalk objects (in the context of a method invocation)

Fig. 1. Annotations currently supported by LogicObjects

3.1 Linguistic Symbiosis

Linguistic symbiosis [3] is the ability of a program to transparently invoke routines defined in another language as if they were defined in its own language [4]. Wuyts and Ducasse [10] add that, to achieve real symbiosis, objects from one language must be understood in the other. In our particular context, these generic symbiosis requirements could be rephrased as being able to:

– Translate Java objects to logic terms, and back.
– Map Java methods to logic queries.

In addition, several problems specific to symbiosis between object-oriented and logic programming languages have been presented in [1,4]. We repeat the most significative from the object-oriented language perspective below:

Unbound Variables: Unlike most object-oriented languages, it is common in logic programming to call a predicate with unbound variables.

Return Values: In object-oriented languages, methods often return objects as a result of their execution. In logic programming, there are no such return values: results are returned by binding values to unbound variables. More than one value can be returned in this way.

Managing Multiplicity: In object-oriented languages there is a difference (e.g., return type) between methods that return a single value or a collection of values. Logic languages make no distinction between predicates that produce a single solution or many solutions.

The expression *"paradigm leak"* [4] has been used in the past to refer to such mapping problems, suggesting a *leakage* of concepts from one paradigm to another. Let us now discuss how our framework deals with these issues.

3.2 Translating Java Objects to Logic Terms

In the context of symbiosis between Java and Prolog, Java objects should have a representation as logic terms and logic terms should be manipulatable as Java objects [10,11]. Since in our technique the first step to map an object to a logic term is to find a mapping between its class and a predicate name, we start by explaining how our framework achieves such a mapping.

Mapping Class Names to Predicate Names. Brichau et al. [11] defined a mapping between class names and predicate names for the specific problem of transforming objects representing parse tree nodes to logic terms and vice-versa. In their work, there is an implicit direct mapping between a logic predicate name and a class name. The arguments of logic predicates are mapped to the children of the parse tree nodes by means of the same recursive algorithm. We generalize their mapping solution by providing, using Java annotations, a general customizable mapping between logic predicate names and Java classes and between predicate arguments and Java object properties.

To illustrate our technique, let us consider the implementation of the `Line` class in Java, shown in Listing 2.6. We refer to this class as a *symbiotic class* since part of its implementation is transparently managed by an object on the logic side. The `@LObject` annotation on line 1 provides custom mapping data for our framework. For example, its optional `name` attribute maps instances of this class to a Logtalk object implementing on the logic side the symbiotic methods of the class. In this case, given that no predicate name is explicitly specified, the name of the corresponding Logtalk object is automatically derived from the class name `Line`. This default mapping is a transformation from Java camel-case naming convention to Prolog names with lowercase tokens separated by underscores. E.g., the Java class `FooBar` would be translated to the Logtalk object `foo_bar`.

This is an example of how, by providing smart default mappings, we reach a complete automation in common cases. At the same time, a programmer can opt for explicitly specifying custom mappings when the defaults are not convenient.

Mapping Java Objects to Logic Terms. When the object on the logic side is a parametric object, its parameters need to be declared on the Java side by means of an `args` attribute in the `@LObject` annotation. In the `Line` class example, this attribute is present in the `@LObject` annotation. It maps the instance variable `name` to the single parameter of the parametric object `line` on the logic side. An instance of the Java class `Line` with its name set to *"central"* is thus automatically translated to the logic term `line(central)`. In this example, the transformation of the object property `name` to a term is straightforward, as it is just a string. If the property had been a symbiotic object (e.g., when its class or a superclass includes an `@LObject` annotation) the transformation process would continue recursively, as the property object could also have properties that are symbiotic objects. When the object on the logic side is not a parametric object, the `@LObject` annotation can be omitted (e.g., the `Metro` class in Listing 2.7).

Mapping Logic Terms to Java Objects. Translating a logic term to an object is the inverse process. However, in this case we need to consider the *translation context*, which encapsulates the translation objective and environment. With this context, we can answer questions such as: Is the translated object going to be assigned to a field? Or is it the result of a symbiotic method (a Java method implemented in Prolog)? Are there relevant annotations in the context (e.g., a field or method) that should influence the translation? What is the expected type of the object in the Java world?

The procedure of transforming a logic term into a Java object starts by attempting to find a symbiotic class whose name and number of parameters correspond to the logic predicate's name and arity. Once we have located and instantiated the logic class, the conversion algorithm continues recursively for mapping each of the term arguments to the object properties. The context provides valuable guidance to choose the right class to instantiate. For example, different Java types could be mapped to a Prolog list representation (e.g., classes implementing the `List` or `Map` interfaces). Therefore, when translating a list term to a Java object, the expected type will influence the selection of the best mapping (e.g., symbiotic classes incompatible with the expected type will be ignored). If many symbiotic classes are compatible with the expected type, by default the framework returns the first match. This can be customized by means of a `preferedClass` attribute in the `LSolution` annotation.

3.3 Mapping Java Methods to Logic Queries

As in [4], methods are mapped by default to logic predicates with the same name and arity. An example of this mapping is found in the `connects(Station, Station)` method (Listing 2.6, line 7). Since this Java method has two parameters, it is mapped to the Logtalk method `connects/2` in Listing 2.2, line 4.

However, a programmer can always customize this mapping by adding a `@LMethod` annotation. The Java method `segments()` illustrates this (Listing 2.6, lines 10–11). As specified by the `name` and `args` annotation attributes, this method will also be mapped to the logic predicate `connects/2`. With this technique, we are thus able to map a single Logtalk predicate, `connects/2`, to different Java methods: `int segments()` and `boolean connects(Station, Station)`, according to our needs. The semantics of these mappings is explained in section 3.6.

3.4 Dealing with Unbound Variables

In Prolog, it is common to write queries with unbound variables. In Java, however, all variables must be bound to a value. Consider the `segments()` method mentioned before. Its arguments are explicitly specified by means of the `args` attribute of the `@LMethod` annotation. These arguments are interpreted as Prolog terms. In this case, both parameters are the symbol "_", which is interpreted as an anonymous logic variable.The class `Station` (Listing 2.5) provides examples of methods having as arguments non-anonymous variables. For instance, the predicate to which the method `connected` (lines 7–8) is mapped, takes as first argument a logic variable *LSolution* and as second argument the first parameter received by the Java method (referred to with the macro expression `$1`).

3.5 Return Values

The result of a logic query can be seen as a set of frames binding logic variables to terms, where each frame corresponds to one logic solution. The solution of a symbiotic method is a transformation from this set of frames to a Java object. By default, a Java object representation of the first logic solution (the first frame) is considered by our framework as the symbiotic method return value. This section discusses techniques for instantiating such Java object from a single logic solution. The composition of a set of solutions is discussed in Section 3.6.

Inferring Return Values from a Logic Variable Name. Our first heuristic is based on a naming convention: If one of the logic variables in a query has as name *LSolution*, its binding in the frame of the first solution will be considered as the term representation of the Java object to return. As an example, reconsider the implementation of the method `connected(Line)` in the `Station` class (Listing 2.5, lines 7–8). This method is mapped to the Logtalk method `connected/2` (Listing 2.3, line 4). As specified by the `args` attribute of the `@LMethod` annotation, the query's first parameter is a Prolog variable *LSolution* and the second parameter is the term representation of the first parameter of the Java method. Upon evaluation of the query, the *LSolution* variable will be bound to a compound term of the form `station(nameStation)`. Given the convention introduced above, the return value of the symbiotic method will be the transformation of this term to a Java object according to the algorithm discussed in section 3.2.

Inferring Return Values from Method Signatures. If no variable with name *LSolution* is found in the query, the framework will attempt to infer its return value from its signature. The term representation of this value has as name the method name (adapted to Prolog naming conventions) and as arguments the parameters of the method. The implementation of the `Metro` class illustrates this. The `line` method (Listing 2.7, line 7) is mapped to a method with the same name on the logic side. In case that the Logtalk method succeeds, the framework will consider as the solution to the method the logic term `line` having as argument the only string parameter of the method. This term will be converted to an instance of the `Line` class according to the algorithm discussed in Section 3.2. In case a line with the name given as a parameter of the Java method does not exist in the logic world, the method will return `null`.

Explicit Specification of Return Values. The previous heuristics reduce the amount of explicit mappings that need to be specified by a programmer. However, we do provide a `@LSolution` annotation to let a programmer specify explicitly the term representation of the Java object to return, overriding the heuristics presented above. This term can be of arbitrary complexity and refer to as many logic variables as required. For instance, if we had wanted to encode explicitly the heuristics for returning the logic variable *LSolution* as the return value of the `connected(Line)` method (Listing 2.5, lines 7–8), we could have done so by annotating it with `@LSolution("LSolution")`. Since this is the default mapping for the solution, it can be omitted, but if an alternative or more complex solution is desired, this can be defined explicitly with the `@LSolution` annotation as well. An example of this is shown in listing 3.1, line 4.

Inferring Return Values from Non-symbiotic Methods. The previous techniques for specifying the return value of a method from a term representation of its result can be generalized to non-symbiotic methods. Methods that should not be mapped to logic routines, but that still want to express their return value as a term expression, can do this by means of the `@LExpression` annotation. For example, Listing 3.1 shows the implementation of a factory class. It provides methods to instantiate certain symbiotic objects that are part of our problem domain. The first method (lines 4–5) creates a new *Station* object by specifying the term representation of its return value with a `@LSolution` annotation. This logic term has the form `station($1)`, where `$1` gets substituted by the first parameter of the method. The second factory method (lines 8–9) does something equivalent to the first one. In this case, no explicit return value is specified with a `@LSolution` annotation, implying that the framework will infer its result from the method signature as discussed before. The term representation of the value to return will be a functor with the same name as the method and having as arguments the method parameters (i.e., `line(String)`).

```
 1 @LObject
 2 public abstract class MetroFactory {
 3     //creates a station with a given name
 4     @LExpression @LSolution("station($1)")
 5     public abstract Station station(String name);
 6
 7     //creates a line with a given name
 8     @LExpression
 9     public abstract Line line(String name);
10 }
```

Listing 3.1. The `MetroFactory` class in Java

3.6 Managing Multiplicity

The previous section illustrated how the framework infers the return value of a symbiotic method from the first solution of a logic routine. This section discusses how to compose a value from multiple solutions, or from properties of the logic solution set.

It is not trivial to infer that a method should return a composition of multiple solutions (e.g., as a list) instead of a single solution. Initially, we tried to infer this from the method return type. For example, if the method returns a collection class, then with certain probability its intention is to return the collection of results rather than a single result. This assumption is not always valid, however. Consider, for example, the method `intermediateStations(Station)` in the `Station` class (Listing 2.5, lines 15–16). This method is mapped to the predicate `reachable/2` in the `station` Logtalk object (Listing 2.3, line 8). The `args` attribute in the `@LMethod` annotation indicates that the first parameter of the Logtalk method will be the logic term representation of the first parameter of the Java method (indicated by the macro-expression `$1`). The second parameter is the Prolog variable *LSolution*. As explained in Section 2, upon execution of the Logtalk method, the *LSolution* variable is bound to a list with the intermediate stations between the receiver station object and the station object passed as first parameter. The method return value is the value bound to that variable in the first solution, according to the heuristics discussed in section 3.5. The Java method thus returns a list of objects that corresponds to the binding of one variable in *one* solution (the first) answered by the Logtalk query. This is an example where a method returning a collection of objects is not intended to answer a single collection of different solutions, but rather a single solution consisting of a collection of objects. In order to resolve ambiguities between both ways of interpreting collections, `LogicObjects` provides the `@LComposition` annotation. The Java method `nearby()` (Listing 2.5, lines 11–12) in class `Station` is an example of the usage of the `@LComposition` annotation (line 11). This method is mapped to the Logtalk method `nearby/1` which takes as argument an unbound logic variable *LSolution*. On the logic side, the unbound variable passed as argument will be bound to a station nearby the receiver station object. On the Java side, as in the previous example, a binding of the *LSolution* variable corresponds to the term representation of an individual solution. Given the `@LComposition` annotation, the framework considers the type of the method (a `List` class) as a container of all its solutions.

Another example is the `lines()` method in the `Metro` class (Listing 2.7, lines 3–4). In this case, the arguments of the method do not include a *LSolution* variable, neither a `@LSolution` annotation. Therefore, the term representation of each solution is given by the name and arguments of the Logtalk method (given explicitly by the `name` and `args` attributes of the `@LMethod` annotation). As in the previous example, the `@LComposition` annotation will instruct the framework to collect all these individual results in a collection. In both cases, the framework will choose a collection class implementing the Java `List` interface, given that this is the return type of the method.

Finally, the return value of a method could be inferred from properties of the complete logic solution set. For example, in case when none of the heuristics discussed in this section can be applied, the framework will inspect the return type of the method. If this is a numeric type, the return value will be the number of results of the query (e.g., the `segments()` method in class `Line`). If it is a boolean, the method answers whether the query produces at least one solution (e.g., the `connects(Station, Station)` method in class `Line`).

3.7 Delegation Objects

We have found cases where the logic representation of a Java object depends on the context where such logic representation is required. To illustrate this, consider a list in Prolog, which is represented as a comma separated list of members as in this example: `[a,b,c]`. In order for the query `[a,b,c]::length(X)` to be valid, two Logtalk objects are required (one object for the empty list, which is an atom, and a parametric object for the non-empty lists, which are compound terms). To maintain a one-to-one mapping, the list methods can be encapsulated in a `list` Logtalk object instead. This allows us to write e.g. `list::length([a,b,c], L)`. On the Java side the best logic representation for a list of objects (e.g., an implementation of `Iterable`) is a logic list term (e.g., `[a,b,c]`). Then this is the default mapping assumed by the framework if nothing else is specified. However, the Logtalk object that knows how to deal with list operations does not correspond to this default logic representation, but rather to the Logtalk object `list`. Therefore, it can be convenient to use this representation in the context of a method invocation. To deal with this kind of situations, our framework provides a `@LDelegationObject` annotation. This annotation allows a programmer to specify mapping data that will be considered in the context of a logic method invocation, and will be ignored in any other context. The class `MyList` (Listing 3.2) shows an example. This class extends the `ArrayList` class, which is translated by default to a logic list term (as all implementations of `Iterable`).

```
@LDelegationObject(name="list", imports="library(types_loader)")
public abstract class MyList extends ArrayList<String> {
    @LMethod(args={"$0", "LSolution"})
    public abstract int length();
}
```

Listing 3.2. A list class declaring a delegation object

The `@LDelegationObject` annotation has the same attributes, with equivalent semantics, as the `@LObject` annotation. In our example, the `name` attribute specifies that the `list` object on the logic side will receive the logic messages sent to instances of this Java class. As we mentioned before, this will not affect the default logic representation of objects that are instances of this class.

The `length()` Java method is mapped to a Logtalk method with the same name. Its first argument is the term representation of an instance of `MyList` receiving the message (referred by the macro `$0`). To build this representation, the framework ignores the `@LDelegationObject` annotation and prefers the default logic representation for lists. The second argument is an unbound logic variable $LSolution$. Upon execution of the logic method, the value bound to this variable in the first solution to the query will become the Java method return value. Given the `@LDelegationObject` annotation, the receiver of the method on the logic side will be the `list` Logtalk object, which provides the method `length/2`, which will bind the second parameter to the length of the list sent as the first parameter.

3.8 Instantiating Symbiotic Classes

To use our framework, a programmer simply needs to instantiate logic classes using a provided factory method. Everything else, including the transparent import of dependencies on the logic side, is automatically managed using runtime code generation and byte code instrumentation techniques. As an example, Listing 3.3 shows an instantiation of a logic class and the invocation of a logic method. The first argument of the factory method corresponds to the logic class to instantiate. The other arguments correspond to the logic class constructor parameters. In this code snippet, the output corresponds to the number of segments on the line *central*, as specified on the logic side (Listing 2.1).

```java
Line line = LogicObjectFactory.getDefault().create(Line.class, "central");
System.out.println("Number of segments: " + line.segments());
```

Listing 3.3. Instantiating a symbiotic class

4 Validation

To validate our approach, we re-implemented with LogicObjects the JPL example of Section 2 (Listing 2.4). This implementation required a significant amount of boilerplate code. Figure 2 shows the notable reduction in code size, and thus in programming effort, that can be gained by using our LogicObjects framework.

The figure also compares the result of a stress test. We show the difference in execution time required by each pair of corresponding methods in the two implementations.[2] Since currently LogicObjects employs JPL to invoke logic routines, the differences in processing time can serve as a measure of the *adaptation effort* (i.e., a measure of the complexity of adaptation heuristics in different scenarios).

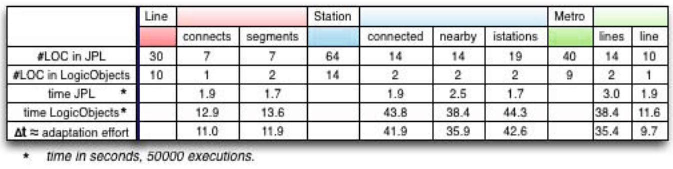
There are many factors that influence such an effort. For example, methods that do not require an adaptation of their parameters (i.e., not including an `args` attribute in a `@LObject` annotation) are the ones with less impact on execution time (e.g., the `connects` method in class `Line` and the `line` method in class `Metro`). On the other hand, methods using macro expressions are among the ones with greater increase in execution time (e.g., the `connected` and `intermediateStations` methods in class `Station`). In addition, the adaptation effort is greater in methods manipulating collection of objects (e.g., the `connected` method in class `Station` and the `lines` method in class `Metro`), since it grows proportional to the amount of objects (also requiring adaptation) in such collections.

In spite of the reduction in program size, the increase in execution time is considerable. However, we regard these results as promising, since there are many optimisation paths to follow in order to reach an acceptable performance in a production setting, such as caching certain mappings so they do not have to be calculated every time, or the usage of a Prolog engine embedded in the JVM. In addition, our framework does not impose any overhead in the execution of a logic routine per se, which is often the real bottleneck performance wise, but on the adaptation of its arguments and the interpretation of its results as objects. For this reason we have preferred to avoid any premature optimisation.

5 Related Work

Several aspects of this work are inspired by SOUL [12], a Prolog dialect that is implemented in and symbiotic with the object-oriented language Smalltalk. Particularly, we improve on the open questions and limitations reported in experiments implementing symbiosis from the object-oriented language perspective. [4]

[2] Tests accomplished with a 2.8 GHz Intel Core 2 Duo processor and 4 GB of RAM.



	Line	🟥	🟥	Station	🟦	🟦	🟦	Metro	🟩	🟩
	🟥	connects	segments	🟦	connected	nearby	istations	🟩	lines	line
#LOC in JPL	30	7	7	64	14	14	19	40	14	10
#LOC in LogicObjects	10	1	2	14	2	2	2	9	2	1
time JPL *		1.9	1.7		1.9	2.5	1.7		3.0	1.9
time LogicObjects*		12.9	13.6		43.8	38.4	44.3		38.4	11.6
Δt ≈ adaptation effort		11.0	11.9		41.9	35.9	42.6		35.4	9.7

* *time in seconds, 50000 executions.*

Fig. 2. A comparison between LogicObjects and JPL

E.g., the reported approach in SOUL for dealing with the problem of returning multiple vs. just a single solution to a query, consists of always returning a collection. When a query has just one solution, its solution is wrapped in a collection wrapper. In order to provide an automatic adaptation, such a wrapper delegates to its wrapped object any message it cannot understand. Unfortunately, this can create subtle problems if the wrapped object is also a collection, as explained in Section 3.6. We have therefore preferred the choice of making logic methods return by default their first solution, and to explicitly use a `@LComposition` annotation whenever the expected return value should be a composition of solutions instead, thus sacrificing a bit on automation to gain on soundness. Concerning how to return values from methods implemented as logic predicates, SOUL limits this answer to the value of a logic variable or an expression written in the object-oriented language. Our approach supports this and in addition allows a programmer to express the value to return as a logic term of arbitrary complexity. We also consider that our technique for mapping method names and their arguments to their logic counterparts is as automatic as the SOUL technique (when relying on the default mappings offered by our framework), without excluding the possibility for other customizations when the defaults do not fit one's needs. Furthermore, our technique for dealing with unbound variables (expressed as annotation arguments) is simpler than the proposal of SOUL of extending the syntax and semantics of the object-oriented language to support the notion of unbound variables on the object-oriented side.

In addition to SOUL, other techniques exist that use advanced linguistic symbiosis for analyzing object-oriented programs (e.g., [13,14]). However, the focus of these techniques is on querying (or transforming) object-oriented programming artefacts from the logic side, rather than achieving an automatic and transparent linguistic symbiosis from the object-oriented language perspective.

There are a number of other works attempting to provide a symbiotic integration between object-oriented languages and Prolog. Most of them do this from the perspective of the logic language, mainly by offering a set of built-in Prolog predicates that enable easy access to the object-oriented language [15,16,17]. In the best cases, libraries for communication from the object-oriented language back to the logic world are provided, but they fail to abstract the programmer from low level mappings, requiring an explicit representation of logic concepts (logic engine, queries, logic terms) in the object-oriented program (as was the case with the JPL example). The same problem occurs for rule engines embedded in Java, like [18], that use a declarative language other than Prolog.

An interesting mapping technique from methods to logic predicates using method type parameters and annotations is presented in [19]. The main shortcoming of this approach is that the types participating in the declaration of symbiotic methods have to be logic term types. Therefore, there is no implicit mapping between objects and their term representations, but term objects must be explicitly created every time a method is invoked.

Another interesting approach that integrates Java with a logic constraint solver is presented in [20]. That work relies on a symbolic virtual machine and the syntax of Java programs is left unmodified. Methods evaluated as logic computations are identified with an annotation. Logic variables are also identified with annotations and are limited to Java primitive types. A limitation is the lack of adaptation of the result of a logic method as in our approach; instead all logic methods must return an object instance of class `Solutions`.

6 Conclusions and Future Work

In this work we presented a framework based on linguistic symbiosis to facilitate the invocation of logic routines from an object-oriented program. The framework focusses in particular on providing a solution from the object-oriented language perspective, since for this direction many difficulties and issues have been reported in the past [4]. Our framework proposes an elegant and customizable solution to each of these previously reported problems. In essence our approach relies on extending the Java language with annotations expressing the declarative nature of certain artefacts, having a counterpart in the logic world.

Although our current implementation is based on Java, most of the ideas presented in this work can be extrapolated to other object-oriented languages. However, we have found that a statically-typed language on the object-oriented side offers considerable advantages for attaining an automatic and transparent symbiosis with a logic language, since valuable information can be extracted from the types of objects belonging to each of the two worlds. This additional type information helps to guide the automatic and transparent conversion of objects from or to the object-oriented world. Nevertheless, in a dynamically-typed language our approach could be reproduced by annotating relevant artefacts (e.g., symbiotic classes and methods) with type data.

Finally, we believe that achieving a complete automatic symbiosis is only possible under the assumption that there is a single valid mapping between artefacts belonging to two different languages. As we have demonstrated, this is not always the case and it is often desirable to let a programmer specify explicitly the desired mapping between artefacts. We have thus opted to provide a "good enough" automation that provides typical default mappings for the most common cases (thus providing a high degree of automation), while at the same time leaving enough flexibility to the programmer to provide more information on the nature and semantics of the desired mapping when required by the problem.

Our future work will focus on implementing a full two-way symbiosis. We plan to use the reflective mechanisms of Logtalk for transparently and automatically referring to Java objects and expressions and invoking their methods in a similar way as has already been accomplished from the Java side. In addition, we will explore techniques for establishing a causal connection between objects belonging to our two different worlds, thus completing a full two-way symbiosis framework.

References

1. D'Hondt, M., Gybels, K., Jonckers, V.: Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis. In: Proceedings of the 2004 Symposium on Applied Computing (SAC), pp. 1328–1335. ACM (2004)
2. Russel, S., Norvig, P.: Artificial Intelligence, A Modern Approach. Prentice Hall (1995)
3. Ichisugi, Y., Matsuoka, S., Yonezawa, A.: RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel. In: International Workshop on New Models for Software Architecture (IMSA): Reflection And Meta-Level Architecture, pp. 24–35 (1992)
4. Gybels, K.: SOUL and Smalltalk - Just Married: Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis. In: Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (2003)
5. Castro, S., Mens, K., Moura, P.: LogicObjects: A Linguistic Symbiosis Approach to Bring the Declarative Power of Prolog to Java. In: Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE (June 2012)
6. Singleton, P.: JPL: A Java Interface to Prolog (September 2012), `http://www.swi-prolog.org/packages/jpl/java_api/index.html`
7. Flach, P.: Simply Logical: Intelligent Reasoning by Example. John Wiley & Sons, Inc., New York (1994)
8. Moura, P.: Logtalk - Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
9. Moura, P.: Programming Patterns for Logtalk Parametric Objects. In: Abreu, S., Seipel, D. (eds.) INAP 2009. LNCS (LNAI), vol. 6547, pp. 52–69. Springer, Heidelberg (2011)
10. Wuyts, R., Ducasse, S.: Symbiotic Reflection between an Object-Oriented and a Logic Programming Language. In: International Workshop on MultiParadigm Programming with Object-Oriented Languages (2001)
11. Brichau, J., De Roover, C., Mens, K.: Open Unification for Program Query Languages. In: Proceedings of the XXVI International Conference of the Chilean Computer Science Society, SCCC 2007 (2007)
12. Wuyts, R.: Declarative Reasoning about the Structure of Object-Oriented Systems. In: Proceedings of the TOOLS USA 1998 Conference, pp. 112–124. IEEE Computer Society Press (1998)
13. De Volder, K.: JQuery: A generic code browser with a declarative configuration language. In: Van Hentenryck, P. (ed.) PADL 2006. LNCS, vol. 3819, pp. 88–102. Springer, Heidelberg (2005)
14. Semmle Ltd.: SemmleCode (2010), `http://semmle.com/`
15. Boulanger, D., Geske, U.: Using Logic Programming in Java Environment (Extended Abstract). Technical Report 10, Knowledge-Based Systems Group, Vienna University of Technology, Austria (1998)

16. Friedrich Bolz, C.: Pyrolog: A Prolog interpreter written in Python using the PyPy translator toolchain, `https://bitbucket.org/cfbolz/pyrolog/`
17. Paul Tarau, P.: Styla: a lightweight Scala-based Prolog interpreter based on a pure object oriented term hierarchy, `http://code.google.com/p/styla/`
18. Friedman-Hill, E.: Jess in Action: Java Rule-based Systems. Manning, Greenwich (2003)
19. Cimadamore, M., Viroli, M.: Integrating Java and Prolog Through Generic Methods and Type Inference. In: Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), pp. 198–205. ACM (2008)
20. Majchrzak, T.A., Kuchen, H.: Logic java: combining object-oriented and logic programming. In: Kuchen, H. (ed.) WFLP 2011. LNCS, vol. 6816, pp. 122–137. Springer, Heidelberg (2011)

A Declarative Compositional Timing Analysis for Multicores Using the Latency-Rate Abstraction

Vítor Rodrigues[2,3], Benny Akesson[4],
Simão Melo de Sousa[1,3], and Mário Florido[2,3]

[1] RELiablE And SEcure Computation Group
Universidade da Beira Interior, Covilhã, Portugal
[2] DCC-Faculty of Science, Universidade do Porto, Portugal
[3] LIACC, Universidade do Porto, Portugal
[4] CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal

Abstract. This paper presents a functional model for timing analysis by abstract interpretation, used for estimation of worst-case execution times (WCET) in multicore architectures using a denotational semantics. The objective aims at surpassing the intrinsic computational complexity of timing analysis of multiple processing units sharing common resources. For this purpose, we propose a novel application of *latency-rate* ($\mathcal{LR}$) servers, phrased in terms of abstract interpretation, to achieve timing compositionality on requests to shared resources. The soundness of the approach is proven with respect to a calculational fixpoint semantics for multicores that is able to express all possible ways in which a shared resource can be accessed. Experimental results show that the loss in precision introduced by the $\mathcal{LR}$ server model is about 10% on average and is fairly compensated by the gain in analysis time, which is above 99%. The system is implemented in Haskell, taking advantages of the declarative features of the language for a simpler and more robust specification of the underlying concepts.

1 Introduction

The timeliness requirement of a software application is defined by the capability of the underlying hardware running the application to assure that execution deadlines are met. In embedded real-time systems, the main timeliness criteria is the worst-case execution time (WCET) of an application [8]. The WCET depends both on the structure of source code, such as loop iterations and function calls, and on hardware factors, such as caches and processor pipelines. In general, the state space of both input data and hardware initial states is too large to be exhaustively explored by measurement approaches. This paper presents a pipeline analysis based on the theory of abstract interpretation [6], aiming at reducing this state space. The design of a proper abstract pipeline domain ensures that the analyzer stabilizes after a finite number of steps over Kleene sequences

[10], while exhibiting a trade-off between the precision of the WCET results and the computational time required by the static analyzer.

When compared to single-core architectures, the complexity of the timing analysis in multicore environments does not depend only on the processor features, but also on the predictability of the timing behavior of each processor when sharing resources, e.g. instruction and data memories [7]. In practice, this means that besides the control flow paths through the program, also the "architectural flows", i.e. the number of ways in which a shared resource can be accessed (also called *interleavings*), must be taken into account. Unless shared resources are shared in a composable manner, the different access interleavings allowed by the scheduling arbiter may produce different intermediate hardware states during analysis and, consequently, affect future timing behavior.

The complexity of the analysis increases exponentially when analyzing architectural flows. Suppose a program that consists of two concurrent processes, P_1 and P_2. The arising conflicts when requesting access to the shared resource are resolved by "interleaving" the execution sequences of the two processes in such a way that either P_1 or P_2 executes by flipping a coin. Hence, for a program with n processes, each one executing a sequence of m instructions, the number of possible interleavings is $(n.m)!/(m!)^n$. The numerator $(n.m)!$ gives all possible interleavings and the denominator $(m!)^n$ restricts this number to the number of allowed sequences, i.e interleavings that preserve the sequential order in the original machine programs. For realistic programs, the number of execution sequences are huge and although their analysis is a decidable problem, it is not feasible to compute in general.

The five technical contributions of this paper are:

1. a static timing analysis for multicore systems, using our previous two-level denotational meta-language [12,14] and an intermediate graph language;
2. the use of the *latency-rate* ($\mathcal{LR}$) server model presented in [17] as an abstraction to achieve compositionality in the temporal domain, so that the analysis of architectural flows can be avoided while preserving the soundness of timing analysis for multicore systems;
3. the formalization and implementation of the $\mathcal{LR}$-server model in the context of data-flow analysis using an abstract interpretation framework based on Galois connections;
4. a method for automatic compilation of dependency graphs, including the new "interleaving" graph operator, into a *meta*-language based on λ-calculus and directly implemented in Haskell;
5. showing that Haskell can be used as a language where the mathematical complex notions underlying the theory of abstract interpretation can be easily, elegantly, and efficiently implemented and applied to the analysis of complex hardware architectures.

The rest of this paper is organized as follows. We start by discussing the related work in pipeline analysis in Section 2, followed by an overview of our approach to the problem of WCET analysis for multicores in Section 3. Section 4 introduces the necessary background on the $\mathcal{LR}$ server model, in particular its ability to abstract timing behavior. Previous work on a two-level denotational meta-language used for static analysis based on abstract interpretation is described in Section 5 and a method to automatically compile fixpoint interpreters using the meta-language is described in Section 6. We then briefly describe our functional approach to a declarative pipeline analysis in Section 7. The formalization of the $\mathcal{LR}$ abstraction in terms of a Galois connection is given in Section 8, followed by a set of Haskell definitions for resource sharing in Section 9. We conclude after a discussion on experimental results in Section 10.

2 Related Work on Pipeline Analysis

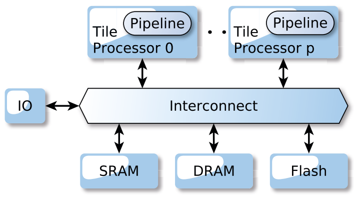
The theoretical foundations of our complete WCET timing analysis framework are the methods of static analysis by abstract interpretation [6] combined with path analysis using linear programming [20]. For the particular case of pipeline analysis, we base our approach on the "abstract pipeline semantics" proposed by Schneider et al. [15], where provably sound timing properties are obtained by abstract interpretation. Since there is no abstraction of sets of *concrete* timing properties, the given pipeline analysis is a special case in the abstract interpretation framework where the abstract timing properties are themselves the "sticky collecting semantics" of concrete timing properties.

The concrete pipeline semantics in [15] is a simplified semantics of the processor, focused only on the aspects related to its temporal behavior, and relies on former value analysis and cache analysis. In contrast, our approach combines value and cache analysis with the pipeline analysis in a single data-flow analysis, which implies that the semantic transformers defined for the register and memory domains can be invoked during pipeline analysis. Still, the theoretical formalism given in [15], in particular its definition of *resource association*, can easily cope with our definition of "hybrid" pipeline state, i.e. a state that combines concrete timing properties with abstract cache states and register invariants.

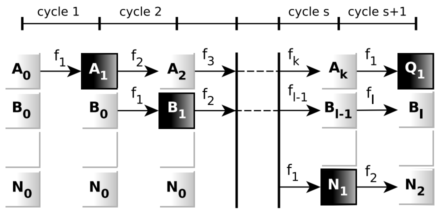
3 Overview of Approach

We consider a tiled multicore architecture with several ARM9 cores, shared memories and IO, as shown in Figure 1(a). Each processor core has an instruction pipeline and an instruction cache memory. By definition, pipelining allows overlapped execution of instructions by dividing the execution of instructions into a sequence of pipeline stages, $k \in PS$, and by simultaneously processing N instructions. We consider a generic ARM9 processor with an instruction cache and a simple in-order pipeline with five pipeline stages: *fetch* (FI), *decode* (DI), *execute* (EX), *memory access* (MEM), and *write back* (WB). Figure 1(b) illustrates a functional view on pipelining.

The functions $f_1, f_2, \ldots, f_k, \ldots$, specify the effect of pipeline state transformations across a variable number of pipeline steps, which is greater than five in the presence of pipeline *bubbles* [15]. For example, the instruction B in Figure 1(b) requires l pipeline steps to complete, where $l > k$. Each pipeline state includes



(a) Generic multicore architecture



(b) Functional overview of pipeline steps

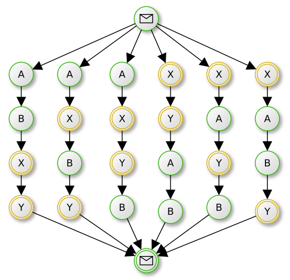
Fig. 1. Functional model of a pipeline in a multicore architecture

an instruction vector of size N, adjoined with a timing property, $1, 2, \ldots, s, s+1$. This property expresses the relation between the elapsed *cycles per instruction* (CPI) and the current stage of an instruction inside the pipeline.

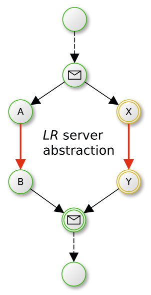
The absence of an abstraction for the *concrete* CPI values in the abstract interpretation literature [15] implies that the *abstract* pipeline domain must be defined as a *set* of pipeline states. For single-core architectures, this does not constitute a computational problem, because there is only a finite, and therefore manageable, number of pipeline states. Although the same principles could apply to timing analysis in multicore architectures, the major drawback is having the sets of concrete timing values spread across a huge number of architectural flows, which is exponentially bigger than the number of control flows.

Let P_1 and P_2 be two processes running on a homogeneous multicore system comprising two processor tiles. The corresponding number of architectural flows is given in Figure 2(a) and the original control flow is given in Figure 2(b).

Assuming composability in the value domain, i.e. there is no application data shared between processes, the need for timing analysis of architectural flows depends on the scheduling made by the arbiter of the shared resource.



(a) Non-compositional timing analysis considering all possible architectural flows between P_1 and P_2



(b) Compositional timing analysis considering only control flows

Fig. 2. Architectural and control flows for two processes P_1 and P_2, where instructions A and B belong to P_1 and instruction X and Y belong to P_2

Composable arbiters, i.e. arbiters providing complete isolation between application in the temporal domain, analysis of interleavings is not required. An example of such an arbiter is non-work-conserving *time-division multiplexing* (TDM), which statically allocates a constant bandwidth to each processor core. However, when replacing the TDM arbiter by a work-conserving *round-robin* arbiter (RR), the system is no longer composable, since the scheduling of requests depend on the presence or absence of requests from other processor cores. In this case, the analysis of every allowed scheduled sequence in Figure 2(a) must be performed.

However, the analysis of the shared resource can be made compositional if the access times are predictable. This implies that upper bounds on the access times to shared resources are calculated so that the variation in interference between processor cores visible in Figure 2(a) is removed (abstracted). The formal model of $\mathcal{LR}$ servers is particularly suitable for determining these upper bounds, since it provides a timing abstraction applicable to most predictable shared resources and arbiters. Figure 2(b) shows how the number of architectural flows can be reduced to the number of control flows when abstracting the temporal behavior by means of the compositional $\mathcal{LR}$-server model.

4 Latency-Rate Servers

We now introduce the concept of latency-rate ($\mathcal{LR}$) [17] servers as a shared-resource abstraction. In essence, a $\mathcal{LR}$ server guarantees a processor core a minimum allocated rate (bandwidth), ρ, after a maximum service latency (interference), Θ. As shown in Figure 3, the provided service is linear and guarantees bounds on the amount of data that can be transferred during any interval independently of the behavior of other processor cores. The values of the two parameters Θ and ρ depend on the choice of arbiter in the class of $\mathcal{LR}$ servers and its configuration. Examples of well-known arbiters in the class are TDM, *weighted round-robin* (WRR), *deficit round-robin* (DRR) and *credit-controlled static-priority* (CCSP) [1].

Like most other service guarantees, the $\mathcal{LR}$ service guarantee is conditional and only applies if the processor core produces enough requests to keep the

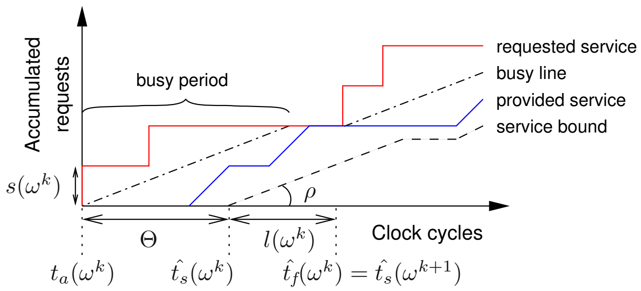


Fig. 3. A $\mathcal{LR}$ server and its associated concepts

server busy. This is captured by the concept of *busy periods*, which are intuitively understood as periods in which a processor core requests at least as much service as it has been allocated (ρ) on average. This is illustrated in Figure 3, where the processor core is busy when the requested service curve is above the dash-dotted reference line with slope ρ that we informally refer to as the *busy line.*

We proceed by showing how scheduling times and finishing times of requests are bounded using the $\mathcal{LR}$ server guarantee. From [19], the worst-case scheduling time, $\hat{t_s}$ of the k^{th} request from a processor core, c, is expressed according to Equation (1), where $t_a(\omega^k)$ is the arrival time of the request and $\hat{t_f}(\omega^{k-1})$ is the worst-case finishing time of the previous request from processor core c. The worst-case finishing time is then bounded by adding the time it takes to finish a scheduled request of size $s(\omega^k)$ at the allocated rate, ρ, of the processor core, which is called the *completion latency* and is defined as $l(\omega^k) = s(\omega^k)/\rho$. This is expressed in Equation (2) and is visualized for request ω^k in Figure 3.

$$\hat{t_s}(\omega^k) = \max(t_a(\omega^k) + \Theta, \hat{t_f}(\omega^{k-1})) \tag{1}$$

$$\hat{t_f}(\omega^k) = \hat{t_s}(\omega^k) + s(\omega^k)/\rho \tag{2}$$

5 Meta-language

This section presents background on our denotational meta-language [12,14]. We develop a constructive fixpoint semantics based on expressions of a two-level denotational meta-language aiming at compositionality in both value and temporal domains. The main advantage is the possibility to generate type safe fixpoint interpreters automatically, and in a flexible way, for a variety of control flow patterns, including the architectural flows originated from shared resources.

Denotational definitions are factored in two stages, which is equivalent to the definition of a *core semantics* at compile-time (ct) and an *abstract interpretation* at run-time (rt). Supported by the *compositionality assumption* of Stoy [18], the core semantics expresses control and architectural flows by means of higher-order relational combinators of the run-time entities.

$$\mathrm{ct} \triangleq \mathrm{ct}_1 * \mathrm{ct}_2 \mid \mathrm{ct}_1 \mathbin{||} \mathrm{ct}_2 \mid \mathrm{ct}_1 \oplus \mathrm{ct}_2 \mid \mathrm{ct}_1 \oslash \mathrm{ct}_2 \mid \text{split rt} \mid \text{merge rt} \mid \mathrm{rt} \tag{3}$$

$$\mathrm{rt} \triangleq \underline{\Sigma} \mid (\underline{\Sigma} \underline{\times} \underline{\Sigma}) \mid \mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2 \tag{4}$$

Implemented combinatores are the sequential composition ($*$), the pseudo-parallel composition ($||$), the intra-procedural recursive composition ($\oplus$), and the inter-procedural recursive composition ($\oslash$). At the compile-time level, we can only directly talk about transformations on run-time values of type $\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2$, defined for program states $\underline{\Sigma}$. These run-time types specify functions that can be regarded as state transformers or simply as "code", whose effect can be obtained by executing a piece of code on an appropriate abstract machine.

Therefore, interpretations of the higher-order expressions of the core semantics (ct) can be used to automatically generate the code of a program (designated by *meta*-program), which is *composed* by several state transformers (rt). The meta-programs are then given as input to the static analyzer. Let $b ::= (\cdot * \cdot) \mid (\cdot \mathbin{||} \cdot) \mid (\cdot \oplus \cdot) \mid (\cdot \oslash \cdot)$ be the syntactical meta-variable for the binary operators in the upper level of the meta-language, and u ::= id | split | merge be the syntactical meta-variable for the unary operators (interface adapters). Then, fixpoints can be generically defined as the reflexive transitive closure $T^\star$ of the transition relation T [4], where T is the initial program relation:

$$T^\star \triangleq \bigsqcup_{n \geqslant 0} T^n = \bigsqcup_{n \geqslant 0} \left(\bigsqcup_{i \leqslant n} T^i \right) = \bigsqcup_{n \geqslant 0} (\lambda R \bullet ((u\, T)\; b\; (u\, R)))^i (\bot_\Sigma) \tag{5}$$

where $\bot_\Sigma$ is the initial hardware state. In this way, fixpoint semantics can be efficiently computed by using program-specific *chaotic iteration strategies* [5], specified at compile-time level by the type expressions in the meta-language for free. In complement to type checking, the soundness of the abstract state transformers, which have the unified type $\mathrm{rt}_1 \underline{\rightarrow} \mathrm{rt}_2$ and are defined at run-time level, can be proven correct by using the calculational approach proposed in [4].

6 Automatic Generation of Fixpoint Interpreters

Next, we describe the calculation process of obtaining fixpoint interpreters. The generation of fixpoint interpreters is based on the notion of relational semantics of the program [3], defined as a set of transition relations $\tau \subseteq (\underline{\Sigma} \times \mathit{Instr} \times \underline{\Sigma})$, where *Instr* is the set of instructions of the program and $\underline{\Sigma}$ is the set of labeled program states according to a weak topological order (w.t.o) [2]. Moreover, the w.t.o. is used to induce a partial *dominance* order $\preceq$ over program instructions.

To represent all the program paths allowed for a program, an intermediate graph language was defined. The inductive abstract syntax of a dependency graph is represented by the data type G and allows us to represent a mimic of the execution order of a program [4], according the program structure known at compile time. The objective is to abstract the trace semantics [3] of the program into a set of "connected" transition relations τ, which are denoted in Haskell by **Rel** a, where a is a polymorphic variable for the domain $\underline{\Sigma}$.

A dependency graph is either an empty graph, a subgraph consisting of a single relation (**Leaf**), two subgraphs connected in sequence (**Seq**), two intra-procedural subgraphs connected recursively (**Unroll**), two inter-procedural subgraphs connected recursively (**Unfold**), two subgraphs connected pseudo-parallely (**Choice**), representing alternative program paths, or, last but not least, two subgraphs running on different processor cores (**Conc**).

```
data Rel a = (a, Instr, a)
data G a = Empty | Leaf (Rel a) | Seq (G a) (G a) | Unroll (G a) (G a)
         | Unfold (G a) (G a) | Choice (Rel a) (G a) (G a) | Conc (G a) (G a)
```

By taking advantage of the algebraic properties of the higher-order relational combinators, fixpoint interpreters (meta-programs) are "calculated" using the denotational approach. The syntactic phrases of a program are their dependency graphs. The denotations of each component of G are expressed by the combinators in the upper-level of the two-level denotational meta-language. The main advantage of using Haskell for the calculation of fixpoint interpreters is the fact that a definition written in Haskell can be compiled (or interpreted) to give a type-safe interpreter. This guarantees the correctness of a core semantics (ct) parameterized by the abstract state transformers defined at run-time (rt).

Along these lines, abstract interpreters in the form of Equation (5) are automatically compiled into λ-calculus by providing interpretations of the core semantics, in particular for the binary operators b and the unary operators u. For example, the sequential combinator $(*)$ is interpreted as the following:

```
( * ) :: (a → b) → (b → c) → (a → c)
(f * g) = λs → (g ∘ f) s
```

The main advantage of defining the higher-order relational combinators in Equation (3) is that new functions can be obtained throughout the composition of more basic functions. In this way, the calculation of meta-programs, all with the unified type $(a \to a)$, is defined by means of the function *derive*. For a complete definition with respect to the patterns in G, we refer to [13].

```
derive :: (a → a) → G a → (a → a)
derive f (Leaf r)  = f * abst r
derive f (Seq a b) = derive (derive f a) b


derive f (Conc a b) = let is = interleavings a b
                          ms = map (derive (create b)) is
                      in f * scatter (length ms) * (distribute ms) * reduce
```

The function *abst* used in the interpretation of the atomic syntactic phrase **Leaf** provides the right-image isomorphism used in the abstraction of the relational semantics to denotation level, as described by Cousot in [3]. By the fact that the structure of dependency graphs is inductive, the type signature of *derive* requires the definition of the actual meta-program f, which is composed in sequence with new interpretations. In this way, the interpretation of (**Seq** a b) is straightforward, stating that the subgraphs a and b are connected in sequence.

The meaning of a subgraph **Conc** a b is given by the composition of the current meta-program f with the whole set of *interleavings* between a and b. The creation and synchronization of these two processes is modeled by the *scatter/reduce* computational pattern, commonly used in parallel computing. Inductively, the derivation of each individual trace is accomplished by using *derive* with the initial meta-program returned by the function *create* [13].

The function *interleavings* is used to obtain the set of architectural flows of Figure 2(a). This function takes two dependency subgraphs and returns a list of subgraphs. Using list comprehensions, the allowed sequences are a subset of all *permutations* of the transition relations belonging to both processes, *main* and *thread*, (which are first converted into lists using *toList*). The illegal sequences are removed by means of the constraint *preserve*, which excludes any sequence *seq*, that, after being filtered from the transition relations belonging to the other process, is not exactly equal to the original sequence *ori*.

```
interleavings :: G a → G a → [G a]
interleavings main thread
   = let preserve ori seq = ori ≡ filter ((flip elem) ori) seq
             (mainL, threadL) = (toList main, toList thread)
             sequences = [is | is ← permutations (mainL ++ threadL),
                                   preserve mainL is, preserve threadL is]
          ts = map traces (groups sequences)
      in map (foldl interleave main) ts
```

After the computation of the interleaved sequences, it is necessary to transform these sequences back into dependency graphs. To this end, the functions *groups*, *traces* and *interleave* are defined according to the logics of the data type G, so that each architectural flow can be instantiated as set of connected transition relations, which possibly pertain to different applications.

In summary, the "derivation" of (**Conc** a b) is first to *scatter* the output state taken from the actual meta-program f into an "array" of independent flows, then *distribute* this state through the array of flows (ms), and finally combine the corresponding outputs using the function *reduce*. The functions *scatter*, *distribute* and *reduce* are described next.

```
scatter :: Int → a → [a]
scatter = replicate

distribute :: [a → a] → [a]     → [a]
distribute = zipWith (λf a → f a)

reduce :: (Lattice a) ⇒ [a] →  a
reduce = foldl join bottom
```

The function *scatter* is trivially defined by the Haskell function *replicate*. The function *distribute* takes a list of functions $[a \to a]$, and a list of input values $[a]$ and return a list $[a]$ with the results obtained by applying each input function to each input value. The function *reduce* is applied at merge points and is responsible for computing the least upper bound between the elements of the input list $[a]$, by using the functions *bottom* and *join* defined in the type class *Lattice* [13].

7 Pipeline Analysis

This section describes our functional and declarative approach to the pipeline analysis of ARM9. The pipeline analysis by abstract interpretation presented in [15] introduces the notion of *resource association* as a pair $(s, \{r_{j_1}, \ldots, r_{j_n}\})$, where $s \in PS$ is a pipeline stage and $r_{j_1}, \ldots, r_{j_n} \in R$ is a set of generic resources, such as functional units or cache memories. These resources can be either static, such as the resource demand of an instruction according to its type, or dynamic, when the description of the resource carries its own state. The particularity in our approach is that the state of the dynamically allocated sequences is updated after each pipeline stage. For this reason, we redefine the notion of a concrete pipeline state in [15] and introduce the notion of a hybrid pipeline state P, which combines concrete timing information with the abstract state of resources.

Let $R^\sharp$ be the abstract register domain, $D^\sharp$ be the abstract data memory domain and $M^\sharp$ be the abstract instruction memory domain. Since the states in $R^\sharp$, $D^\sharp$ and $M^\sharp$ can be updated during every pipeline stage and need to be shared by all instructions inside the pipeline, we require the definition of an extra set of store buffers $R'^\sharp$, $D'^\sharp$ and $M'^\sharp$. These domains contain the resource states that are to be allocated during the pipelining of every single instruction. This means that, after analyzing an instruction, it is required to compute the least upper bound between the top-level domains $R^\sharp$, $D^\sharp$ and $M^\sharp$ and the store buffers $R'^\sharp$, $D'^\sharp$ and $M'^\sharp$. The hybrid pipeline state is defined as:

$$P \triangleq (\mathit{Time} \times \mathit{Pc} \times \mathit{Demand} \times R'^\sharp \times D'^\sharp \times M'^\sharp \times \mathit{Coord}) \tag{6}$$

where *Time* is the global number of CPU cycles, *Pc* is *program counter* of the next instruction to fetch, *Demand* is a 32-bit sized word, used to model the dependencies between data registers in such a way that each register is either a blocked or unblocked resource, and *Coord* is a N-sized vector, N being the number of instructions allowed inside the pipeline at a given time.

$$\mathit{Coord} \triangleq [\mathit{TimedTask}]_N \tag{7}$$

A *TimedTask* is defined for one instruction and consists of the current elapsed CPU *Cycles* and the current *Stage* of a given *Task*. A *Task* is associated with an instruction, *Instr*, and holds also local copies of the "context" of a hybrid state:

$$\mathit{TimedTask} \triangleq (\mathit{Cycles} \times \mathit{Stage} \times \mathit{Task}) \tag{8}$$

$$\mathit{Task} \triangleq (\mathit{Instr} \times \mathit{Pc} \times \mathit{Demand} \times R'^\sharp \times D'^\sharp \times M'^\sharp) \tag{9}$$

We now identify semantic transformers required by our *functional approach* to pipeline analysis, as illustrated in Figure 1(b). The analysis is performed at three levels: at the lower level, we define the transformer F_T as a morphism on the composite domain *TimedTask* (for example, the instances $f_1, f_2, \ldots, f_n$ in Figure 1(b)); at the middle level, we define the transformer F_P as a morphism on the composite domain P, which uses F_T to compute the new elements inside the N-sized vector *Coord*; finally, at the higher level, we define the transformer $F_P^\sharp$ as a morphism on sets of hybrid states $P^\sharp \triangleq 2^P$, which uses F_P to transform the hybrid pipeline states in the input set. The semantic transformers F_P and $F_P^\sharp$ are concisely defined as:

$$F_P \in \mathit{Instr} \mapsto P \mapsto P \tag{10}$$

$$F_P(i)(p) \triangleq \mathit{toContext}(i) \circ [F_T \circ \mathit{fromContext}(p)]_N \tag{11}$$

$$F_P^\sharp \in \mathit{Instr} \mapsto P^\sharp \mapsto P^\sharp \tag{12}$$

$$F_P^\sharp(i)(p^\sharp) \triangleq \{F_P^{5+}(i)(p) \mid p \in p^\sharp\} \tag{13}$$

where F_P^{5+} corresponds to the recursive functional application of F_P at least five times in an ARM9 pipeline. Note that F_P^{5+} does not correspond to the transitive closure of F_P by the fact that *local* worst-case timing properties are always associated with the final pipeline stage of a given task. This is possible because the value and cache analysis are performed simultaneously with the pipeline analysis, thus making the timing analysis a deterministic process for each given input timing property. In this way, the intermediate hybrid pipeline states can be discarded after completion.

However, even in fully timing compositional architectures [7], such as ARM9, the non-determinism introduced by the control flow must be taken into account. Therefore, the soundness can only guaranteed if all hybrid pipeline states arriving at a join point are collected into a set of type $P^\sharp$. The definition of $F_P^\sharp$ naturally supports the non-determinism intrinsic to sets of hybrid states in the sense that

F_P^{5+} is applied to every pipeline state $p \in P^\sharp$. Let $\{s_k^i \mid k \in PS, k \geqslant 5\}$ be the set of ordered pipeline stages (including stalled stages) required to complete the instruction i. Then, F_P^{5+} is defined by:

$$F_P^{s_{k+1}^i}(i)(p) \triangleq F_P(i)(F_P^{s_k^i}(i)(p)) \tag{14}$$

$$F_P^{5+} \triangleq F_P^{s_{WB}^i} \tag{15}$$

The purpose of F_T is to compute the effect of pipelining a single instruction. However, since all the N instructions inside the coordinate vector (*Coord*) share the common context defined in P, it is necessary to read/write the state of the resources in P. In particular, the value of the program counter Pc must be known to fetch the next instruction from memory when one instruction inside the pipeline finishes, and the value of *Demand* must be kept updated depending on the blocked/unblocked state of register ports.

As an example, consider the case where the current stage is **FI** (*Fetch*), i.e. there is free space inside the pipeline to fetch a new instruction from instruction memory. Depending on the context of the actual pipeline state P, structural hazards [15] may block the access to memory and, therefore, cause the pipeline to stall. Otherwise, the actual *TimedTask* is updated by means of the function *fetchInstr*, which uses the context information about the next program counter to fetch *pc* and the actual state of the abstract instruction memory *iMem* to calculate the output timing property. Here, a timing property is denoted by the type variable a implementing the type class *Cycles*, as defined by Equation (8).

```
fetchInstr :: (Cycles a) ⇒ a → Task → TimedTask a
fetchInstr cycles t@Task {taskNextPc = pc, taskImem = iMem}
   = let (classification, opcode, m′) = getAbstMem iMem pc
         i = decode opcode
         pc′ = pc + 4
         buffer′ = setAbstReg bottom R15 (StdVal pc′)
     in if classification ≡ Hit
         then let t′ = t {taskInstr = i, taskNextPc = pc′, taskImem = m′}
              in TimedTask {property = fetched cycles, stage = DI,
                            task = Fetched t′ buffer′}
         else  let t′ = t {taskInstr = i, taskNextPc = pc′, taskImem = m′}
              in TimedTask {property = missed p, stage = FI,
                            task = Stalled Structural t′ buffer′}
```

Two different scenarios can occur during a fetch: either the *opcode* of the instruction is contained in the instruction cache, in which case the memory access is classified as a **Hit** and the next stage is set to **DI** (*Decode*); or it must be fetched from instruction main memory, thus causing the pipeline to become **Stalled**. In any case, the abstract cache state is updated by means of function *getAbstMem*. The type class *Cycles a* defines two functions, *fetched* and *missed*, for each of the corresponding scenarios. If the instruction fetch was successful, then the abstract value of the register **R15** (the program counter register in ARM9) is updated in the store buffer using *setAbstReg*. Due to page limitations, the reader can find the complete Haskell definition of the pipeline analysis in [13] (see Section 9 for another example of the use of declarative programming).

8 The $\mathcal{LR}$-Server Model as a Galois Connection

The meaning of the access times to shared resources in the context of timing analysis is the range of its possible values, i.e. the interval from lower bounds to upper bounds. Due to the limited bandwidth of the shared bus, shared accesses introduce additional delays that stall the pipeline. Therefore, the soundness of the timing analysis requires the computation of upper bounds on delays. To cope with this, we redefine *TimedTask* as:

$$\mathit{TimedTask} \triangleq (\mathit{Cycles} \times \mathit{Delay} \times \mathit{Stage} \times \mathit{Task}) \tag{16}$$

As mentioned in Section 7, the pipeline abstract domain is defined as a set of hybrid pipeline states, each including a "concrete" timing property now given by *Cycles* plus *Delay*. The purpose of the $\mathcal{LR}$-server model is to reduce the number of joins and provide, at the same time, upper bounds for delays caused by shared requests. From the observation of Figure 2(a), it is clear that the number of join operations is proportional to the number of architectural flows. However, Figure 2(b) shows that when applying the $\mathcal{LR}$ model to compute *safe* upper bounds for the finishing times of shared requests, the number of joins is determined solely by the control flows of each process independently.

The soundness of the abstraction provided by the $\mathcal{LR}$-server model relies on the fact the all timing properties calculated throughout architectural flows are upper bounded by the finishing times calculated using the $\mathcal{LR}$ model. Here, the objective is to formalize this approximation by means of a Galois connection.

Let *Delay* be an upper semi-lattice equipped with a partial order $\leqslant$ on natural numbers $\mathbb{N}$, describing both concrete and abstract timing properties and let $\mathbb{D}$ be a set of timing properties. A Galois connection $\mathit{Delay}^{\natural}(\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathit{Delay}^{\sharp}(\subseteq)$, where $\mathit{Delay}^{\natural} = \mathit{Delay}^{\sharp} = 2^{\mathbb{D}}$, is defined in terms of a *representation function* $\beta : \mathit{Delay} \mapsto \mathbb{D}$ that maps a concrete value $p \in \mathit{Delay}$ to the best property describing it in $\mathbb{D}$. This property is the canonical extension of Equation (2) to sets. Given a subset $X \subseteq \mathbb{D}$ and an abstract property $p^{\sharp} \in \mathit{Delay}^{\sharp}$, the abstraction and concretization maps are defined by:

$$\alpha(X) = \bigcup\{\beta(x) \mid x \in X\} \tag{17}$$

$$\gamma(p^{\sharp}) = \{p \in P \mid \beta(p) \subseteq p^{\sharp}\} \tag{18}$$

Let w_c^k be the k^{th} instruction to fetch from the shared memory when there is a cache miss in the processor core c. The best property $p^{\sharp}$ is the singleton set containing the smallest finishing time given by Equation (2) when applied to w_c^k. Therefore, the $\mathcal{LR}$ abstraction can be formally defined by the representation function β:

$$\beta(t_f(w_c^k)) = \{\max(t_a(w_c^k) + \Theta_c, \hat{t}_f(w_c^{k-1})) + s(w_c^k)/\rho_c\} = \{\hat{t}_f(w_c^k)\} \tag{19}$$

This formally shows that the predictability of $\mathcal{LR}$ servers can be used to abstract the meta-programs corresponding to architectural flows into meta-programs corresponding to control flows only. Since each access time is upper bounded by the $\mathcal{LR}$ server, we have by compositionality that the maximum local timing property given by Equation (17), that would be obtained by joining ($\bigcup$) all abstract pipeline states across the architectural flows in Figure 2(a), is exactly equal to the maximum local timing property when only the control flows are considered.

9 Haskell Definitions for Resource Sharing

This section gives declarative definitions for the temporal behavior of TDM and $\mathcal{LR}$ arbiters. Let the type variable *a*, defined in the type class *Cycles a*, be instantiated by a concrete timing property denoted by the data type *WCET*.

```
data WCET = WCET {cycles :: Int, ta :: Int, core :: Int, tf :: Int, delay :: Int}
```

The analysis of a TDM arbiter is simplified due to its predictable and composable properties, which makes the delay of a request to a shared resource easily computed using the arrival time, *ta*, and the processor *core* identifier. As mentioned in Section 7, requests to the main instruction memory occur upon cache misses. Thus, the function *missed* belonging to the type class *Cycles* is:

```
missed w@WCET {cycles = c, ta, core}
  = let d = ta `mod` frame
        first = slots * core
        end = first + slots - 1
        ts = if first ⩽ d ∧ d ⩽ end then 0
               else if d < first then (first - d) else (frame - d + first)
    in w {cycles = c + round (ts + 1), tf = ta + ts + 1, delay = ts + 1}
```

The frame size of the TDM bus is given by the variable *frame*. Assuming slots are equally distributed among the processor cores and that they are consecutively allocated in the frame and a completion latency of 1 cycle, the delay time is *ts+1*, where *ts* uses the division remainder of the arrival time *ta* by *frame* in order to check for an allocated slot. If the *core* needs to wait for an allocated slot, the required number of cycles can be statically calculated [9].

Now consider a shared bus with an arbitration protocol that is predictable but not composable, such as work-conserving round robin. In this case, the timing behavior of each application is dependent on the applications running on other cores, which makes analysis of all architectural flows mandatory in order to achieve soundness. In this context, the advantage of the $\mathcal{LR}$-server abstraction is the possibility to guarantee bounds on the starting times and finishing times of the requests so that compositionality in the timing domain is achieved.

The $\mathcal{LR}$-server model requires a timing property to model the guaranteed service rate, which is the finishing time (*tf*) of the previous request on the same *core*. According to Equation (2), the function *missed* defines the timing behavior of a cache miss in terms of an arrival time, *ta*, and a previous finish time, *tf*. Accordingly, the function *missed* is:

```
missed w@WCET {cycles = c, ta, tf = tf'}
  = let busy = ta + theta < tf'
        d = if busy then 1/rho else theta + (1/rho)
    in w {cycles = c + round d, tf = d + if busy then tf' else ta, delay = d}
```

10 Experimental Results

The discussion of experimental results include two different experimental scenarios. First, we compare the WCET and the analysis time obtained for small programs from the analysis of architectural flows (TDM) versus control flows (TDM) in Table 1. Second, we compare the WCET results of composable TDM versus a $\mathcal{LR}$ abstraction of a composable TDM arbiter for Mälardalen WCET benchmark programs [11] in Table 2. By compositionality of the $\mathcal{LR}$ abstraction and assuming that each processor core has a sufficiently large private data memory (D-\$) and a common initial hardware state, each program is analyzed independently from the program configured to run on the second core. We consider the simplified multicore architecture in Figure 4(b), where instructions are shared in a partitioned SRAM memory shared by a TDM arbiter.

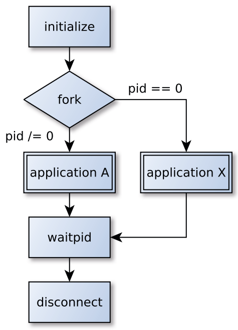
By definition, architectural flows cannot be feasibly computed. However, we do compute interleavings for the simple program in Figure 4(a), where "application A" and "application X" have only a few instructions each. Due to its natural composability, the analysis of control flows with TDM arbitration is much faster than the analysis of architectural flows, requiring only 1% of the time. With respect to the WCET estimate, the first line in Table 1 shows a lower WCET (179 CPU cycles) for the interleavings approach compared to composable TDM analysis (185 CPU cycles). This difference in the WCET is a consequence of the actual hardware state of the processor core running "application X" upon the invocation of the *fork* procedure and demonstrates the impact that the intermediate hardware states have on the timing analysis of architectural flows.

In fact, when the number of instructions of "application X" is bigger than the number of instructions of "application A", the worst-case path corresponds to that of "application X". However, since the analysis of "application X" starts with an empty pipeline state, it naturally takes less CPU cycles to complete. After increasing the number of instructions in "application A", this effect is eliminated because the worst-case path becomes that of "application A". Consequently, for the two analyses, the WCET is equal in the last two experiments.

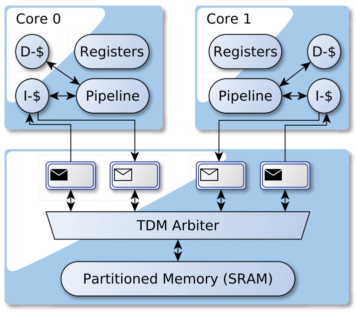
Table 1. Comparison results for architectural flows, composable TDM

No. instructions "application A"	No. instructions "application X"	No. of interleavings	Results (CPU cycles/sec.)	Architectural Flows (TDM)	Composable TDM
4	5	126	WCET	179	185
			Analysis Time	57.0	0.17
5	5	252	WCET	188	188
			Analysis Time	140.3	0.18
6	5	462	WCET	195	195
			Analysis Time	588.7	0.43

Next, we compare the WCET results in Table 2 obtained using the $\mathcal{LR}$ abstraction with $\Theta = 1$ and $\rho = 0.5$ (modeling a particular TDM configuration with frame size of 2) to the results obtained with composable TDM. The WCET



(a) Example of a multi-process program



(b) Simplified multicore architecture

Fig. 4. Simple program running on a simplified multicore architecture

values presented in Table 2 depend not only on the size of the instruction cache and on the ability of the $\mathcal{LR}$ server to stay busy, but also on the program flow, e.g. number of loop iterations. Since we are considering a blocking multicore architecture, where a request from a processor core cannot be issued before the previous request has been served, every request starts a new busy period by definition. This is the most unfavorable situation possible for the $\mathcal{LR}$ abstraction, since every request requires $\Theta + 1/\rho$ cycles to complete, maximizing the overhead compared to TDM.

Still, our experiments show that this overhead is limited to between 8.7% and 12.1% for the considered arbiter, configuration, and applications. This is partly because the use of a small frame size reduces the penalty of starting a new busy period upon every cache miss through the low $\Theta = 1$ value, but also because the case of an SRAM shared by a TDM arbiter is quite simple and is captured well by the abstraction. A more complex case with DRAM and CCSP arbitration is shown in [16] along with an optimization to reduce the pessimism of the abstraction without loss of generality. In terms of the run-time of the analysis tool, it is approximately ($\approx$) the same for both composable TDM and the $\mathcal{LR}$ abstraction.

From this experiment, we conclude that compositional analysis of control flows using the $\mathcal{LR}$ abstraction is very fast and scalable compared to analysis of architectural flows. The analysis time is similar to compositional analysis based on composable TDM arbitration, although it incurs a reduction in accuracy of about 8-12% for our configuration and applications. More precise WCET estimates would be obtained for multicore architectures that support high levels of parallelism. For example, architectures including super-scalar pipelines or caches allowing multiple outstanding requests. This would reduce the number of busy periods in the $\mathcal{LR}$ server upon cache misses, but would also increase the overall complexity of the WCET analyzer. Nevertheless, the main benefit of the $\mathcal{LR}$ abstraction is that it is able to perform compositional timing analysis using any arbiter belonging to the class, as opposed to being limited to composable TDM.

Table 2. WCET results for some of the Mälardalen benchmarks

Benchmark	No. Source Loop Iterations	$\mathcal{LR}$-server (WCET)	No. Cache Misses	TDM (WCET)	Overhead (%)	Analysis Time in sec. (≈)
bs	152	1162	111	1036	10.8	2.3
bsort	156	1459	152	1311	10.1	0.9
cnt	145	1309	175	1171	10.5	0.8
cover	111	796	105	707	11.2	3.9
crc	459	3160	304	2826	10.6	15.0
expint	251	2023	233	1818	10.1	1.9
fdct	1011	10897	720	9892	9.2	20.1
fibcall	111	994	59	885	11.0	2.3
matmult	287	2580	188	2343	9.2	5.2
minmax	221	956	263	873	8.7	2.6
prime	232	1079	196	959	11.1	5.2
ud	418	3943	97	3464	12.1	40.0

11 Final Remarks

This paper presents an approach to timing analysis in multicore architectures exclusively based on the declarative frameworks of denotational semantics, abstract interpretation and functional programming. The type system of Haskell is used to define a type safe and parameterizable fixpoint semantics by means of a two-level denotational meta-language. Fixpoint (abstract)-interpreters are automatically generated by providing interpretations to the algebraic combinators of the meta-language, providing a generic and compositional framework for static analysis. A particular abstract interpreter for pipeline analysis is defined for the WCET analysis of programs running on the ARM9 microprocessor.

The WCET analysis of multicores is defined incrementally by extending the intermediate representation language with a new syntactical element, representing programs running on different processing cores, whose denotational interpretation reuses the algebraic combinators used for static analysis in single cores. The complexity of the new fixpoint interpreter is reduced by using the abstraction provided by the $\mathcal{LR}$ server model on the timing behavior of shared resources.

Using declarative programming in Haskell, the temporal behavior of shared resources is in direct correspondence with the mathematical definitions of the TDM and $\mathcal{LR}$ arbiter models. The outcome is the definition of provably sound and compositional timing analysis in multicore environments, with a loss of precision in order of 10% on average that is relatively small compared to the factor 100 reduction in terms of analysis time.

Acknowledgments. This work is partially funded by LIACC, through the Programa de Financiamento Plurianual, FCT, by the FAVAS project, PTDC/EIA-CCO/105034/ 2008, FCT, and by the EU ARTEMIS JU funding, within the RECOMP project, ref. ARTEMIS/0202/2009, JU grant nr. 100202.

References

1. Akesson, B., Hansson, A., Goossens, K.: Composable resource sharing based on latency-rate servers. In: Proc. DSD (2009)
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. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Elec. Notes in Theoretical Computer Science, 6 (1997)
4. 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)
5. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13(2-3) (1992)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL (1977)
7. Cullmann, C., Ferdinand, C., Gebhard, G., Grund, D., Maiza, C., Reineke, J., Triquet, B., Wilhelm, R.: Predictability considerations in the design of multi-core embedded systems. In: Proc. ERTS (2010)
8. Engblom, J., Ermedahl, A., Stappert, F.: A worst-case execution-time analysis tool prototype for embedded real-time systems. In: Proc. RT-TOOLS (2001)
9. Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., Roychoudhury, A.: Bus-aware multicore wcet analysis through tdma offset bounds. In: Proc. ECRTS (2011)
10. Kleene, S.C.: Introduction to metamathematics. Van Nostrand (1952)
11. Mälardalen WCET research group, `http://www.mrtc.mdh.se/projects/wcet`
12. Rodrigues, V., Pedroso, J.P., Florido, M., de Sousa, S.M.: Certifying execution time. In: Peña, R., van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2011. LNCS, vol. 7177, pp. 108–125. Springer, Heidelberg (2012)
13. Rodrigues, V.: A declarative compositional timing analysis for multicores using the latency-rate abstraction. Technical report, LIACC, Faculty of Computer Science, University of Porto (2012), `http://www.dcc.fc.up.pt/~vitor.rodrigues`
14. Rodrigues, V., Florido, M., de Sousa, S.M.: A functional approach to worst-case execution time analysis. In: Kuchen, H. (ed.) WFLP 2011. LNCS, vol. 6816, pp. 86–103. Springer, Heidelberg (2011)
15. Schneider, J., Ferdinand, C.: Pipeline behavior prediction for superscalar processors by abstract interpretation. ACM SIGPLAN Not. 34 (1999)
16. Shah, H., Knoll, A., Akesson, B.: Bounding SDRAM Interference: Detailed Analysis vs. Latency-Rate Analysis. In: Proc. DATE, pp. 308–313 (2013)
17. Stiliadis, D., Varma, A.: Latency-rate servers: a general model for analysis of traffic scheduling algorithms. IEEE/ACM T. Netw. 6(5) (1998)
18. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press (1977)
19. Wiggers, M., Bekooij, M., Smit, G.: Modelling run-time arbitration by latency-rate servers in dataflow graphs. In: Proc. SCOPES, pp. 11–22 (2007)
20. Wilhelm, R.: Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 309–322. Springer, Heidelberg (2004)

Supporting Pruning in Tabled LP[⋆]

Pablo Chico de Guzmán[1], Manuel Carro[2,3], and Manuel V. Hermenegildo[2,3]

[1] ElasticBox, Inc.
[2] IMDEA Software Institute, Spain
[3] School of Computer Science, Univ. Politécnica de Madrid, Spain
pchico@clip.dia.fi.upm.es, {mcarro,herme}@fi.upm.es

Abstract. This paper analyzes issues which appear when supporting pruning operators in tabled LP. A version of the once/1 control predicate tailored for tabled predicates is presented, and an implementation analyzed and evaluated. Using once/1 with answer-on-demand strategies makes it possible to avoid computing unneeded solutions for problems which can benefit from tabled LP but in which only a single solution is needed, such as model checking and planning. The proposed version of once/1 is also directly applicable to the efficient implementation of other optimizations, such as early completion, cut-fail loops (to, e.g., prune at the toplevel), if-then-else, and constraint-based branch-and-bound optimization. Although once/1 still presents open issues such as dependencies of tabled solutions on program history, our experimental evaluation confirms that it provides an arbitrarily large efficiency improvement in several application areas.

Keywords: Logic Programming, Tabling, Pruning, Performance.

1 Introduction

Tabled LP [1] overcomes several limitations of the SLD resolution strategy. In particular, it guarantees termination for programs with the *bounded term-size* property and can improve efficiency in programs which repeatedly perform some computation. These characteristics help make logic programs less dependent on clause and goal order, thereby bringing operational and declarative semantics closer together. Tabled LP has been successfully applied in many areas including deductive databases, program analysis, or semantic Web reasoning.

The operational semantics of tabled LP differentiates the first call to a tabled predicate, the *generator*, from subsequent variant calls (calls which are identical modulo variable renaming), the *consumers*. Generators resolve against program clauses and insert the answers they compute in the *table space*. Consumers read answers from the table space and suspend when no more answers are available (therefore breaking infinite loops) and wait for the generation of more answers by their generators. A generator is said to be *complete* when it is known not to be able to generate more (unseen) answers. In order to check this property, a fixpoint procedure is executed where all the

[⋆] Work partially funded by MINECO project TIN-2008-05624 *DOVES* and CAM project S2009TIC-1465 *PROMETIDOS*.

consumers inside the generator execution subtree are reading their pending answers until no more answers are generated. After completion, memory used by consumer suspensions can be reclaimed. The completion operation is complex because a number of generators may be mutually dependent, thus forming a Strongly Connected Component (SCC [2]) in the graph of generator dependencies. As new answers for any generator can result in the production of new answers for any other generator of the SCC, we can only complete all generators in an SCC at once, when the completion fixpoint has been reached. The SCC is represented by the *leader* generator: the youngest generator which does not depend on older generators. A leader generator defines the next completion point.

One of the key decisions in the implementation of tabled LP is when a generator returns its computed answers, i.e., the *scheduling* or *evaluation strategy*. *Local evaluation* is the most widely spread evaluation strategy: it executes the full completion fixpoint procedure before returning any answer outside the generator execution subtree. It is efficient in terms of time and stack usage when all answers are needed, but performs speculative work when only a subset of the answers is required. The speculative work performed by local evaluation makes pruning quite ineffective in practice, since it cannot take place until all answers have already been computed.

A work-around for the speculative work of local evaluation is *answer-on-demand* tabled evaluation, where generators return answers as soon as they are computed. The first attempt proposed is *batched evaluation*, but it can be very inefficient memory-wise because it delays completing fixpoint computations without reclaiming the memory used by consumer suspensions. *Swapping evaluation* [3] works around this issue with a memory behavior which is closer to that of local evaluation. Swapping evaluation avoids speculative work before returning demanded answers, but it performs the same amount of work as local evaluation when backtracking. This brings the necessity for pruning operators in tabled LP in order to be able to discard unnecessary alternative execution paths. The contribution of this paper is a *discussion of the issues related to pruning in tabled LP which motivate the implementation of an efficient pruning operator —a version of* once/1*— with a more natural semantics for the realm of tabling than that of the standard cut operator.* once/1 *is implemented under swapping tabled evaluation, and we identify a series of optimizations, programming patterns, and general types of applications where it can be used advantageously*. The final goal is to enlarge the domains in which tabled LP can be put to work in a natural way.

This paper concentrates on *proper tabling* (or *suspension-based tabling*), which does not recompute execution paths in order to recover the execution state of a suspended consumer. Also, *variant tabling* is assumed, i.e., a call is considered a consumer iff it is identical, modulo variable renaming, to a previous generator. Adapting the proposed solutions to work under *subsumptive tabling*, which considers a goal A to be a consumer of a goal B if $B\theta \equiv A$ for some substitution θ (or, in general, for tabling under constraints where consumers are defined under the notion of entailment [4]) is left for future work. We assume some familiarity with the WAM [5] and proper tabling implementations.

The rest of the paper is organized as follows: Section 2 introduces a number of issues that appear when performing pruning in tabled LP and proposes solutions for them. Section 3 motivates the use of pruning operators in tabled LP by showing different

```
:- table t/2.

t(A,B) :- p(A,C), !, ...
t(A,B) :- ...

p(A,B) :- t(B,C), ...

?-t(X,Y).
```

Fig. 1. !/0 example

```
:- table t/1.

t(X) :- t(Y), !, fail.
t(1).

?-t(X) vs ?- t(1).
```

Fig. 2. !/0 inconsistency

```
:- table t/1, r/1.

t(X) :- r(X).
t(1).

r(X) :- t(X).
r(2).

?-once(t(X)) vs ?-once(r(Y)),once(t(X)).
```

Fig. 3. Solution order dependency

applications where this combination is useful. Section 4 shows some implementation details of the proposed once/1 pruning operator. Section 5 evaluates our solution experimentally. Finally, Section 6 gives an overview of the related work and Section 7 summarizes some conclusions.

2 Issues to Support Pruning in Tabled LP

A desirable feature of any program language is "to compute only what is needed and to compute it only once". Tabled LP is useful to solve the second problem, but it needs the combination of answer-on-demand tabled evaluation with pruning operators to solve the first one. This section analyzes the issues of combining tabled LP with pruning operators, showing the drawbacks of the standard cut operator (!/0) and proposing a different declarative semantics for our version of the once/1 pruning operator.

2.1 !/0 Operator in Tabled LP

The operational semantics of !/0 is strongly based on the depth-first search strategy of Prolog. The fact that tabled LP does not follow this strategy — the execution order of tabled clauses is dynamic — makes the operational semantics of !/0 under Prolog to be not applicable under tabled evaluation. This is specially relevant in the presence of mutually recursive calls (Figure 1). It is for example quite possible that !/0 cuts the second clause for one call to t/2 (when, e.g., t(*B*,*C*) is a consumer of a complete tabled call) but not for other calls (when, e.g., t(*B*,*C*) is a consumer which suspends). This behavior is caused by the effects of !/0 spanning across clauses. This is inadequate in the context of tabled LP, since the execution order of the clauses of a generator is not always easy to predict.

Another example of the ill behavior of !/0 in tabled LP is shown in Figure 2. The tabled evaluation of the query t(*X*) executes the first clause of t/1 and suspends the consumer call t(*Y*), executing the second clause of t/1 on backtracking. The second clause of t/1 generates the answer *X*=1, which is returned to the toplevel. On backtracking, *X*=1 is consumed by the consumer t(*Y*) before the final failure of the execution. On the other hand, the tabled evaluation of the query t(1) executes the first clause of t/1, but now t(Y) does not suspend. t(Y) can be either a generator — which would return answers after its evaluation — or a consumer which reads the available t(1) answer. Thereby, t(Y) succeeds and !/0 prunes the second alternative before executing a failure. Therefore, !/0 produces an inconsistency since t(1) fails and t(X) succeeds with the answer $X = 1$.

2.2 Behavior of Once/1

As we have seen, !/0 adapts badly to tabled LP, but pruning is a necessity for general program techniques such as generate-and-test programs if the generation of further potential solutions is to be pruned when a test condition succeeds — i.e., if only one solution (a *witness*) is necessary. In the context of tabled LP, once/1 provides a much more appropriate semantics: once(G) executes G but it also cancels any external backtracking over G. once(G) is guaranteed to produce at most one solution without any guarantee as to which particular solution it is — it could even be a random pick — and can be expressed in terms of !/0 for a non-tabled goal as once(G) :– call(G), !. Thereby, once/1 is less dependent on the execution order of the generator clauses than !/0 because once/1 does not span across clauses.

once/1 is also useful in order to extend the functionality of !/0 for updating tabling data structures. For example, consider a once(G) call which succeeds. The data and control structures which would be necessary to re-enter the execution of G are not needed any more. To this end, once/1 must remove not only the choicepoints belonging to the current execution path — as !/0 does — but also the consumers which appeared inside the execution subtree of G. The resumption of these consumers might lead to subsequent solutions of the once/1 call, which would contradict the previous rationale. Note that these consumers are, in implementations that use proper tabling, either protected from backtracking or copied away to a separate memory area and would not be pruned by !/0. Note also that this consumer removal, which is necessary for correctness, is not done by other tabled LP approaches to pruning and is not trivial. For example, one of these consumers, say C, might belong to the consumer list of a generator not being pruned, and the completion fixpoint operation of this other generator would resume C.

One may expect that once/1 will return the solution which is less expensive to compute (e.g., the first one to be computed), but the solution order in tabled LP depends on the shape of generator dependencies, which in turn depends on the program history in classical tabled LP implementations. Consider the program in Figure 3 and the query once(t(X)). Execution enters the first clause of t/1 and calls the generator r(X). The execution of the first clause of r/1 suspends since t(X) is a consumer. r(X) computes the solution $X = 2$ and this is the only solution propagated to the toplevel because of the once/1 success. On the other hand, the execution of the query once(r(Y)), once(t(X)) calls the generator r(Y) and its first clause calls to the generator t(Y), whose first clause suspends because r(Y) is a consumer. t(Y) computes the solution $Y = 1$. After that, once(t(X)) is called, which can consume the previous solution $X = 1$. This is the only solution propagated to the toplevel because of the success of once/1. Therefore, the solution to once(t(X)) depends on the execution history. Moreover, if we impose $X = 1$ after calling once(t(X)), the execution succeeds or fails depending on the program history. We can resume this behavior in the following dependency chain:

program history ⇒ SCCs ⇒ solution order ⇒ once/1 solution ⇒ program results

The shape of generator dependencies could be made to depend on statically predictable characteristics which would remove dependencies from the history, but we did not find any completely satisfactory order. For example, the lexicographical order could be used, but let us consider the execution of the previous query once(t(X)). When t(X) is called from the first alternative of r(X), t(X) could be recomputed as if it was a generator because r(X) cannot depend on t(X) (r(X) comes first under the lexicographical order). Given a program, its solutions would not depend on the program history since the lexicographical generator priority fixes the shape of generator dependencies, but a change to the names of the tabled predicates could change the order of the solutions, which is arguably not the best situation. In general, orders which rely on syntactic characteristics of the source code are sensitive to changes on the program text which are very common and which programmers do not expect to result in alterations to the behavior of the program.

This dependency on the program history is an open issue in existing tabled LP supporting pruning operators, although it has not been documented before. However, we strongly believe that having a once/1 operator available is worthwhile because the combination of pruning and answer on-demand tabled evaluation is very efficient in a variety of applications, as we will show in the next section. Also, the behavior of the program execution in tabled LP with pruning operators is consistent as long as the programmer is aware that (s)he cannot rely on which particular solution will be returned by a call to once/1. Keeping this property in mind, the query once(t(X)), $X = 1$ makes little sense and it is a questionable programming pattern.

3 Applications of Once/1

We motivated the once/1 operator as a general instrument for programs which benefit from tabled LP but which only need a subset of all the possible solutions. In addition, there are a number of programming patterns where once/1 is quite useful and which are worth mentioning.

3.1 Generate and Test Applications

Consider a model checker based on tabled LP, such as XMC [6], which performs reachibility analysis. A typical case is the verification of a mutual exclusion protocol where each configuration state is a tuple with a state q_i for each process P_i. For example, for three processes the state $\langle q_1, q_2, q_{me} \rangle$ represents a configuration where process P_1 is in state q_1, process P_2 is in state q_2, and process P_3 is in a *mutual exclusion* state.

A model checker application would provide the predicate reach/2, which returns in its second argument all the configuration states reachable from the configuration state given by its first argument. Therefore, all the configuration states reachable from the state I_0 are returned by the query ?- reach(I_0,X). Note that, for verification purposes, the search can be stopped when two processes are in the mutual exclusion state at the same time. This condition can be expressed with the following facts and query (where initial/1 returns the initial state):

```
check(⟨$q_{me}$,$q_{me}$,_⟩).
check(⟨$q_{me}$,_,$q_{me}$⟩).
check(⟨_,$q_{me}$,$q_{me}$⟩).

?- initial($I_0$), once(reach($I_0$,$X$), check($X$)).
```

3.2 Early Completion Optimization

Early completion [7] is an optimization for tabled LP which completes a generator call when a new answer does not further instantiate the call and is therefore the most general answer. In that case, further backtracking over the early-completed generator is unnecessary. This is the same objective that a once/1 which succeeds pursues. Early completion optimization can then be easily implemented by associating a once/1 call which does not appear in the program and whose final activation is to be dynamically decided (which we term a *virtual* once/1) with all the generator calls. When all the free variables of a generator call remain unbound when one of the generator answers is found, a (virtual) success of the generator virtual once/1 call can be simulated. As we will see in Section 5.2, early completion optimization based on once/1 clearly outperforms other early completion optimization implementations. Also, as early completion optimization is performed when free variables remain uninstantiated, early completion optimization based on once/1 does not present the issues commented in Section 2.2.

3.3 Pruning at the Top Level

A (virtual) once/1 call can be also associated to the toplevel query in order to perform pruning at the top level. Similarly to the implementation of early completion optimization, pruning at the top level when no more answers are demanded by the user can be achieved by simulating a (virtual) success of the toplevel virtual once/1 call.

3.4 If-Then-Else Prolog Transformation

The Prolog program transformation for the classical *Cond* -> *A*;*B* statement is as follows:

```
if-then-else(Cond,A,B)) :- Cond, !, A.
if-then-else(_,_,B)) :- B.
```

which does not work if *Cond* needs tabled evaluation. This is due to two main reasons: a) *Cond* might suspend and then, *B* would be executed; later on, a resumption of *Cond* might lead to the execution of *A*; b) as remarked in Section 2, !/0 does not ensure at most one solution of pruned tabled calls. The first issue can be solved by supporting negation in tabled LP [8], which is usually implemented by the tnot/1 operator. The second one can be solved by using once/1 instead of !/0. The new transformation for *if-then-else* statements in tabled LP would be:[1]

```
if-then-else(Cond,A,B)) :- once(Cond), A.
if-then-else(Cond,_,B)) :- tnot(Cond), B.
```

[1] Note that the call to tnot/1 succeeds at most once.

```
:- table path/3.
path(X,Y,Cost) :-                 min_path(X,Y,Best,Min) :-
     edge(X,Y,Cost).                  once(path(X,Y,Best)),
path(X,Y,C) :-                        NewBest #< Best,
    C #= C1 + C2,                     once((min_path(X,Y,NewBest,Min) ; Min = Best)).
    C #>= 0,
    edge(X,Z,C1),                 ?- min_path(X,Y,_,Min).
    path(Z,Y,C2).
```

Fig. 4. Constraint-based optimization

3.5 Application to Minimization Problems

Although we currently support only variant tabling under swapping evaluation, Ciao can combine tabled LP and CLP [4] and work to combine them with once/1 under swapping evaluation is underway. The resulting system can be applied, for example, to a declarative and efficient formulation of optimization. Consider the program in Figure 4.[2] min_path/4 iteratively calls path(X,Y,$Cost$) and successively constrains the path cost. It is called inside once/1 because we are interested in a single solution. Note that the recursive calls can perform the reactivation operation (which will be explained in Section 4.6) in order to continue the generator execution at the point where it was pruned after imposing some more tighter constraints. When the constraints are too tight and the path/3 call fails, the immediately previous cost is returned. The procedure implements a *branch and bound* algorithm where tabled LP avoids loops and redundant work, constraints are used to implement bounds which cut the search, and once/1 restricts the search to return only one witness.

4 Implementation Details of the once/1 Operator

This section recalls the general ideas of swapping evaluation [3] and explains the implementation of the once/1 operator, which is based on the management of once scopes and the pruning procedure associated with them. We will also see some optimizations as the reactivation operation or memory reclaiming after a pruning operation.

4.1 Swapping Evaluation

Pruning needs answer-on-demand tabled evaluation to be more effective. Swapping evaluation [3] is an answer-on-demand strategy for tabled LP which solves the memory consumption issues of batched evaluation. It implements a different behavior for *internal* and *external* consumers. An internal consumer appears inside the execution subtree of the leader of the generator of the consumer. E.g., in a program with clauses {(:−table a/0), (a :− a), (a)} with the query ?− a, a, the leftmost a/0 in the query is a generator, the a/0 in the body of the first clause is an internal consumer, and the rightmost

[2] The symbol # differentiates constraints from arithmetical operators.

a/0 in the query is an external consumer. Using swapping evaluation, internal consumers behave as usual, but external consumers read answers from the table space and, when no more answers are available, they move the choice points and their corresponding trail cells of their generators to the top of the stacks in order to modify the backtracking execution order. The original generator is then transformed into a consumer and the external consumer becomes a generator which can produce more answers, avoiding the use of memory for external consumer suspensions — which is the most important source of memory consumption in batched execution.

4.2 Once Scope Data Structure

A *once scope* is a data structure associated with a once/1 call which keeps track of relevant information in order to perform the pruning operation. Once scopes are hierarchically organized, because a once/1 call can be called from the execution subtree of another once/1 call (the latter being the *parent* of the former). Note that this hierarchical structure includes the (virtual) once scopes associated to generator calls. Therefore, the consumer list of a generator is directly accessible via its virtual once scope.[3]

A once scope S is composed of the following fields: choicepoint, parent, children set, consumer set and generator set. choicepoint indicates the choicepoint at time of the once/1 call corresponding to S. parent indicates the parent once scope of S. children set stores the set of once/1 calls which are immediately called from S (those once scopes whose parent field points to S). consumer set is the set of consumer calls which are called when S is the *active once scope*. The active once scope is the youngest once scope of those whose execution subtree is being executed. Similarly, generator set is the set of generator calls which are called when S is the active once scope.

4.3 The Management of Once Scopes

Figure 5 shows Prolog code for the once/1 operator, which is responsible for managing the once scopes. Once scopes are stored on the *once scope stack*, whose topmost element is the *active once scope*. new_once/1 initializes $Scope$, a new scope for the current once/1 call. It initializes the choicepoint and parent fields of $Scope$ and updates its children set field.[4] push_once/1 pushes $Scope$ onto the once scope stack to indicate that it is now the active once scope. If G succeeds, once_proceed/0 performs the pruning operation related to the active once scope. After that, pop_once/0 pops off $Scope$.

```
once(G) :-
    new_once(Scope)
    push_once(Scope),
    undo(forward_trail(
        push_once(Scope),
        pop_once)),
    call(G),
    once_proceed,
    pop_once,
    undo(forward_trail(
        pop_once,
        push_once(Scope))).
```

Fig. 5. once/1 predicate

One additional difficulty is that consumer calls which suspend within a once/1 call have discontiguous executions. Let us consider the call once(C), where C is a consumer

[3] We consider the consumer list of a generator as the consumers appearing inside the generator execution subtree instead of the repeated calls up to variable renaming. This does not affect the completion operation fixpoint.

[4] children set is also updated when a generator completes or after a swapping operation.

call which suspends. The execution might exit the once/1 subtree on suspension and reenters it when resuming, which requires the once scope structure to be popped off on suspension and pushed on on resumption. The mechanism we have used is to leave actions on the trail to be executed on untrailing (e.g., when suspending to enter another clause) and on resumption (when reinstalling the trail to continue a suspended call after a new answer is available). We insert in the trail, via the undo/1 operation,[5] the forward_trail/2 goal, which is defined to execute its second argument when called. This second argument is then invoked on backtracking when C is suspended. The resumption mechanism in turn recognizes forward_trail/2 when reinstalling the trail and calls its first argument. Therefore, the first undo/1 always discards the scope when the call finally fails. In the case of a consumer inside the execution subtree of once/1, it also uninstalls the scope when performing untrailing to suspend and pushes the scope back onto the stack on resumption. The second undo/1 performs the reverse operation, which is needed to neutralize the actions of the first undo/1 in order to resume consumers outside the execution subtree of the current once/1 call: it reinstalls the once scope on backtracking which will be popped off by the first undo/1 and pops off the once scope which has been previously reinstalled by the first undo/1 on consumer resumption.

The virtual once scopes associated with generators and the toplevel execution are managed by a similar code, but once_proceed/0 is not executed by default. For these cases, once_proceed/0 is executed if the early completion optimization can be performed or no more answers are demanded by the user, respectively.

4.4 Terminology

Note that the consumers and generators of a once scope S also belong to the once scope of the parent of S (although they do not directly appear in their consumer/generator set fields). We recursively define the *once-recursive consumer set* of a once scope S as the members of the consumer set field of S plus the *once-recursive consumer set* of the members of the children set field of S. We define similarly the *once-recursive generator set* of a once scope.

Remember that we have associated a *virtual* once scope to all generators. The consumer list of a generator — those appearing inside its execution subtree but not in the execution subtree of internal generator calls — are the members of the once-recursive consumer set of the once scope associated with the generator. We also define the *recursive consumer set* of a once scope S as the once-recursive consumer set of S plus the *recursive consumer set* of the once scopes associated with the members of the once-recursive generator set of S, i.e., it also includes the consumers inside the execution subtree of internal generators, and therefore it is made up of the set of consumers inside the execution subtree of the once/1 call. We also define the *recursive generator set* of a once scope S accordingly.

For example, following Figure 6, there is a once/1 call (associated to the once scope $ONCE_B$) which is internal to the execution of another once/1 call (associated to the once scope $ONCE_A$). $ONCE_{G_1}$ and $ONCE_{G_2}$ are the virtual once scopes associated to, respectively, the generators G_1 and G_2. These generators are called from the execution

[5] undo/1 is a common facility which leaves a goal call in the trail to be invoked on untrailing.

$ONCE_A$:
Parent: $NULL$
Consumer Set: $\{C1, C2\}$
Generator Set: $\{G1\}$
Once Set: $\{ONCE_B\}$

$ONCE_B$:
Parent: $ONCE_A$
Consumer Set: $\{C3\}$
Generator Set: {}
Once Set: {}

$ONCE_{G_1}$:
Parent: $ONCE_A$
Consumer Set: $\{C4\}$
Generator Set: $\{G2\}$
Once Set: {}

$ONCE_{G_2}$:
Parent: $ONCE_{G_1}$
Consumer Set: $\{C5\}$
Generator Set: {}
Once Set: {}

Fig. 6. Once structures

of the once/1 call associated to $ONCE_A$. The internal consumers of a virtual once scope are included into the recursive consumer set of any of its parent once scopes, but they are not included into the once-recursive consumer set of any of its parent calls. Consequently, the once-recursive consumers of the once scope $ONCE_A$ are $C1$, $C2$ and $C3$, and the recursive consumers of the once scope $ONCE_A$ are $C1$, $C2$, $C3$, $C4$ and $C5$.

4.5 The Pruning of a Once Scope

once_proceed/0 is responsible for pruning the active once scope. Its pseudo-code is:

```
DELETE $act\_once\_scope$ from ParentOf($act\_once\_scope$);
$current\_choicepoint$ = InitChoicepoint($act\_once\_scope$);
for each $G \in$ recur_gen_set($act\_once\_scope$) do state($G$) = PRUNED;
```

The first line deletes the active once scope from the once scope set of its parent once scope. This operation causes the removal of all the consumers inside the execution subtree of the active once scope, because the active once scope (and its consumers) is not reachable from the once scope of any generator any more and then, these consumers will not be traversed by the execution of any completion fixpoint procedure. After that, the current choicepoint is updated to the one of the active once scope in order to discard pending search of the execution subtree of the once/1 call being pruned. Finally, all the non-complete generators inside the execution subtree of the active once scope are marked as PRUNED in order to avoid inconsistencies if one of their consumers appear. We follow a similar approach to the one of incomplete tables [9], but it is improved with the use of the *reactivation operation* (see Section 4.6). The main goal of the incomplete table proposal is to avoid the generator recomputation when the answers of a PRUNED generator are enough to evaluate a (future) consumer. (Future) consumers consume the available answers from its PRUNED table, and only if all such answers are exhausted, the generator is computed from scratch. Later, if the computation is pruned again, the same process is repeated until eventually the subgoal is completely evaluated. Note that each recomputation from scratch computes at least one more solution, keeping the tabled LP termination property for programs with the *bounded term-size*.

```
:- table t/1, r/1.

t(X) :- r(X).
t(1).
...
t(X) :- ...

r(X) :- large_comp, once(t(X)).
```

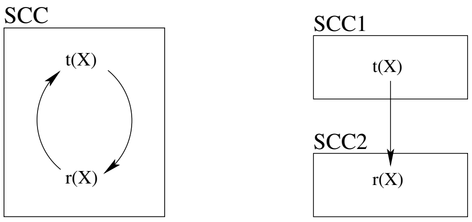


Fig. 7. Consumer Optimizations

4.6 Pruning Optimizations

We here propose some optimization which can be applied to the implementation of once/1. They do not affect to the operational semantics of once/1, but can improve the time/memory execution of tabled LP applications.

The Reactivation Operation. The subtree under a pruned generator might not be fully explored at the moment of pruning, possibly discarding the computation of pending answers. Thereby, (future) consumers of pruned generators might require answers which were not computed due to the pruning. Two main approaches have been proposed so far: either keep the solutions in the answer table and protect the execution subtree from backtracking [10], or keep the solutions in the answer tables but discarding the execution subtree [9]. The former, based on the *reactivation operation*, might be interesting for applications where the pruned generators are often reactivated, arbitrarily improving the execution speed. However, we decided to implement the latter because the memory consumption of the former could be unacceptable. For example, using stack freezing, the trail section to be saved at time of pruning is unbounded because backtracking might be performed until the initial choicepoint.

We improve over incomplete tables for cases where the reactivation operation comes for free. once_proceed/0 marks as PRUNED the generators inside the execution subtree of the active once scope and updates the current choice point, but these operations can be performed lazily on backtracking (just before entering the pruned alternatives). Therefore, the execution subtree of pruned generators is kept on the stacks and can be reused by a swapping operation executed in the continuation code. This special case of the swapping operation implements the reactivation operation for free. On the other hand, after backtracking over the prune generators, the execution of the pruned generators is reclaimed and it must be computed from scratch if future consumers demand more answers than the ones available.

Memory Optimizations after Pruning Success. Another optimization is related to the removal of the consumers inside the execution subtree of active once scope — which trivially speeds up the completion operation because fewer consumers have to be traversed in the completion operation fixpoint. This removal can be also used to reduce dependencies between generators being evaluated — fewer dependencies lead to the sooner completion of generators and better memory use — and to reclaim unneeded consumer memory of pruned consumers which is protected from backtracking.

In the example in Figure 7 (left-side), the query t(X) is a generator whose first clause calls r(X), another generator. r(X) starts a large computation and, afterward, once(t(X)) is called. The first clause of t(X) suspends when calling r(X). The generator dependency graph at this moment is shown in Figure 7 (in the middle), where there is only a completion point represented by the leader t(X). After the consumer call to r(X) suspends, execution backtracks and the second clause of t(X) is executed, computing the answer X=1. This makes the once/1 call succeed and the previously suspended consumer is removed (and therefore ignored by the completion fixpoint). This can be used for reclaiming the memory associated to these consumers —which is probably frozen on the stacks —and for updating the graph of generator dependencies as shown in Figure 7 (right-side), removing the dependency of r(X) on t(X). The new graph of generator dependencies defines two different completion points, corresponding to two different leaders, t(X) and r(X). Therefore, r(X) can complete on backtracking. The completion of r(X) improves the program memory behavior because the memory used by large_comp/0 is reclaimed before exploring alternative clauses of the generator t(X). The pseudo-code to perform this optimization is as follows:[6]

```
$Oldest_{leader}$ = oldest(Gen($C$) s. t. ($C \in$ recur_cons_set($act\_once\_scope$)));
**for** ($G = G_a$; $Oldest_{leader}$ != $G$; $G$ = ParentOf($G$))
  Leader($G$) = oldest($G \cup$ {Gen($C$) s. t. ($C \in$ recur_cons_set(OnceScope($G$)))});
```

The first line computes the oldest dependency of the active once scope. The second line traverses generators starting from the youngest one being executed, G_a, until $Oldest_{leader}$, in order to update their generator dependencies.[7] Since once_proceed/0 has just been executed, the third line computes the new oldest dependency of G without taking into account the pruned consumers. At the end of the execution of this code, the leader fields have been updated according to the new graph of generator dependencies.

There is another optimization for reclaiming the memory frozen by consumers which is independent from the updating of generator dependencies. This optimization refers to the case where the topmost frozen memory corresponds to consumers being pruned. In this case, we can update the value of the frozen memory in order not to protect from backtracking the memory of consumers which have been pruned. The pseudo-code for this optimization is as follows:[8]

```
$MAX_{froz\_mem}$ = max(FrozMem($C$) s. t. ($C \in$ recur_cons_set($act\_once\_scope$)));
**if** ($FrozMem$ == $MAX_{froz\_mem}$)
  $FrozMem$ = max(FrozMem($C$) s. t. ($C \in$ recur_cons_set(OnceScope(Leader($G_a$)))));
```

The first line computes MAX_{froz_mem}, the maximum frozen memory by the consumers inside the execution tree of the once scope being pruned. Since once_proceed/0

[6] This code follows the call to once_proceed/0 in Figure 5.

[7] Generator dependencies of generators older than $Oldest_{leader}$ are unaffected.

[8] This code follows the call to once_proceed/0 in Figure 5.

has just been executed, the third line computes the new maximum frozen memory without taking into account the pruned consumers. $FrozMem$ is only updated if its previous value is different than the one of MAX_{froz_mem} because if these values are the same, the current frozen memory corresponds to a consumer outside of the execution subtree of the once scope being pruned.

5 Performance Evaluation

We have implemented the once/1 pruning operator under swapping evaluation in Ciao and compared its performance w.r.t. XSB version 3.3.6. Both systems were compiled with gcc 4.5.2 and executed on a machine with Ubuntu 11.04 and a 2.7GHz Intel Core i7 processor.

5.1 Applications Searching an Answer Subset

Table 1 shows execution times in ms. for a set of applications which can take advantage of once/1 in order to compute a subset of answers. numbers searches for an arithmetic expression which evaluates to a given natural number (N), given a list of natural numbers (S) (100+ lines). Tabled LP is used to avoid the recomputation of recursive calls with a subset of S. The suffix none indicates a query where N cannot be obtained using S, the suffix easy indicates a query where the first solution implies the computation of a small fraction of the search space and suffix stand indicates a query with no special characteristics (i.e., no specific search tree shape was sought). iproto, leader, and sieve are model checking applications where reachability analysis is performed (600+ lines each). numbers uses once/1 in the definition of its tabled predicates in order to return only one answer. iproto, leader and sieve queries are embedded in a once/1 operator in order to prune the search when the first answer is returned, e. g. once(iproto(init,$FinalState$)). We measure the time to return the first answer for each query in the *first* column and also until final failure the *all* column (i.e., when all the solutions are computed, when that is the case). We show execution times of Ciao under local and swapping evaluation, using once/1 or not.

numbers_none cannot take advantage of either swapping evaluation or once/1 because it must explore the full search space, since no solutions are found. Its different execution times provide an intuition regarding the overhead of swapping evaluation and once/1, which in this case are both almost negligible. In numbers_easy, local evaluation has to compute all the possible expressions while swapping evaluation can return the first one and stop, which takes much shorter. The use of once/1 allows swapping evaluation to discard alternative execution paths before performing backtracking — note that swapping evaluation would compute all the answers on backtracking unless once/1 is used, which gives us a strong reason for the necessity of combining answer-on-demand tabled evaluation and pruning operators. With respect to local evaluation, it takes some more time to return all the solutions than to return the first one, because they have to be reconstructed from the table space where they were stored after having been computed before returning the first one. once/1 under local evaluation makes it possible to discard solutions when the first one is found, but recursive tabled calls were still completely evaluated in a speculative way. We conclude that a pruning operator can be used

Table 1. Execution time (ms.) of local vs. swapping evaluation with/without pruning operators

	Local				Swapping			
	No once/1		once/1		No once/1		once/1	
Query	first	all	first	all	first	all	first	all
num_none	21 912	21 912	22 365	22 365	22 574	22 574	22 982	22 982
num_easy	20 613	21 108	17 023	17 027	1 624	22 421	1 708	1 792
num_stand	24 296	25 312	22 750	22 753	8 403	26 058	8 729	8 906
iproto	2 992	3 184	3 014	3 016	1 112	3 024	1 126	1 141
leader	9 940	10 324	10 182	10 186	3 296	10 963	3 412	3 433
sieve	35 554	36 272	35 986	35 991	7 081	37 139	7 107	7 123

under local evaluation, but it is obviously less interesting than under swapping evaluation. numbers_stand has a behavior similar to numbers_easy, but the first solution takes some more time to be computed. Note that the effects of pruning on execution time depend heavily on when the first answer is found, because pruning only affects the remaining search space. iproto, leader, and sieve show an overall behavior similar to that of numbers.

5.2 Early Completion Based on Once/1

Existing proposals for early completion optimization are highly dependent on the syntactic form of the generator clauses and often allow unnecessary computations. For example, the XSB early completion optimization updates the next instruction of the generator choicepoint to be the completion fixpoint procedure, avoiding the computation of the alternative generator clauses. It does not perform either reactivation of pruned generator calls or updating of the graph of generator dependencies based on the consumer removal. These drawbacks are overcome by our early completion optimization based on once/1. As an example, let us analyze the behavior of the handcrafted code in Figure 8 in XSB. t1/0 is a generator whose first clause calls t2/0, another generator. t2/0 calls t1/0, performing a consumer suspension. On backtracking, t2/0 cannot complete because of this dependency. Now, the second clause of t1/0 is executed and it succeeds. At this moment, t1/0 can be completed early (discarding pending execution alternatives), but its fixpoint procedure is still executed. The consumer of t1/0 is resumed, (speculatively) executing a large computation (sleep(2)). Obviously, the resumption of this consumer is unnecessary for forward execution and it would not have been performed under early completion optimization based on once/1. In contrast, the generator t2/0 would have been marked as pruned to be later reactivated if needed.

Table 2 shows execution times in ms. for a set of benchmarks which can take advantage of early completion optimization. genome computes relations following a genome structure represented as a graph. The suffixes give some rough indications of the shape of the graph. We measure Ciao and XSB, using local evaluation in both cases for fairness in the comparison.[9] The no_early column shows execution times taken after modifying the XSB sources to deactivate early completion optimization and the early column

[9] Note that early completion is effective even under local evaluation since it prunes the generator execution after computing its first solution.

Table 2. Execution time (ms.) of early completion optimization

	XSB			Ciao		
	no_early	early	speedup	no_early	early	speedup
bad_xsb	2000	2000	1.00	2000	0.3	6666
genome_chain	28.8	24.8	1.16	33.1	23.5	1.41
genome_grid	102.4	60.4	1.69	116.2	12.7	9.15
genome_cycle	290.0	212.5	1.36	324.7	1.8	180.38
genome_dense	3.2	0.4	8.00	5.1	0.2	25.50

shows execution times with the early completion optimization activated. As explained previously, the early completion optimization in Ciao is based on once/1.

bad_xsb takes 2000 ms. in XSB and less than 1 ms. in Ciao, confirming the previous analysis. The rest of the benchmarks show more realistic scenarios, where XSB (no_early) executes sometimes faster than Ciao (no_early). One reason is that the Ciao tabled LP implementation is based on a program transformation which imposes some overheads. But the main focus of interest here is the search space which is pruned by the early completion optimization. XSB takes advantage of early completion optimization, speeding up the execution between 1.16× and 8×, while Ciao obtains speedups between 1.41× and 180×, showing that early completion optimization based on once/1 can clearly be more effective in many cases. The execution times of genome_cycle and genome_grid, which generate situations similar to the one of bad_xsb where the Ciao early completion optimization discards expensive fixpoint computations which XSB executes, are the ones where Ciao gets the most advantage w.r.t. XSB.

```
:- table t1/0, t2/0.

t1 :- t2.
t1.

t2 :- t1, sleep(2).

?- t1.
```

Fig. 8. bad_xsb example

6 Related Work

To the best of our knowledge, there are five previous attempts to incorporate pruning operators in tabled LP [11,12,9,13,10]. [11] works under the dynamic reordering of alternatives (DRA) technique [14]. Because the abstract machine for DRA is much more WAM-like than the implementations of proper tabling, the authors claim that the DRA implementation of cut is closer to that of !/0 in the WAM. They argue that, since the DRA scheduling strategy is deterministic, this allows for a well-defined !/0, with a more intuitive operational semantics. DRA tries non-looping alternatives first and looping alternatives later on, and this is the order in which !/0 prunes. In fact, proper tabling implementations could be made to follow the same order for consumer resumptions as DRA. However, we tend to agree with [10] that the behavior resulting from the implementation of !/0 in DRA can still be confusing, as argued in Section 2.2. Also, DRA is based on recomputation of looping alternatives, while proper tabling does not re-execute except for the cost of reinstalling trailed bindings, offering a quite different trade-off. Our proposal is tailored to proper tabling.

Tabling modes [12] is also based on !/0. It is used at the level of program definition, which restricts the flexibility for the case of applications which sometimes need all the solutions and sometimes need a subset of them. It uses a lazy strategy, which computes all the solutions as local evaluation does. Consequently, tabling modes do not prune the tabled evaluation. A minimization problem as that in Section 3.5 would not use previous solutions to prune the search space.

Incomplete tables [9] is also based on !/0. They do not provide a robust implementation (Yap Prolog documentation alerts that the behavior of tabled LP with !/0 is undefined). Also, its implementation does not support the reactivation operation.

Demanded tables [13] implements a version of once/1. In this work, calls which are being consumed by external consumers (demanded table) are not pruned, which makes it necessary to perform runtime analysis to detect if a generator call is being demanded. We avoid this analysis by supporting reactivation of tables. We do not care if a generator to be pruned is being demanded, since the demanding consumers would reactivate the generator if needed.

JET [10] is closer to the spirit of this work, although no implementation is provided. The ideas presented are also based on reactivation of tables, but this work does not provide any pruning operator for the user. Instead, pruning takes place on *JET points*, which are detected by static analysis. This is a deliberate design decision to facilitate the job of the programmer, but it implies a loss of pruning power. For example, our numbers benchmark would not benefit from JET pruning. We strongly believe that the semantics of once/1 is clear enough for the programmer, although we could of course adapt our pruning operator to be based on analysis. Other minor advantages of our pruning operator are that once/1 is linear in the number of generator choicepoints while JET pruning is linear in the number of choicepoints, that once/1 does not impose any overhead if pruning is not used, and that once/1 does not store any choicepoint more than once to allow future reactivations, among others.

Finally, the most important contribution of our pruning mechanism is the pruning of consumers inside the execution subtree of a pruning operator. They must be removed in order not to execute the continuation of a pruning operator more than once — the resumption of these consumers might lead to a new execution of this continuation code. Moreover, we propose some memory optimizations to take advantage of the consumer removal after a pruning operation.

7 Conclusions

We argue that none of the previous approaches for pruning in tabled LP is fully satisfactory although a pruning operator under answer-on-demand tabled evaluation is a necessity in order to enlarge the application domain of tabled LP. To this end, we have presented and evaluated a pruning operator under swapping evaluation, and reported on benchmarking of its implementation in Ciao, comparing it to previous proposals, and showing that it offers advantages in terms of efficiency and programmability. We have also shown how our pruning operator can be used as a basis for implementing a number of optimizations.

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43(1), 20–74 (1996)
2. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM J. Comput. 1, 140–160 (1972)
3. Chico de Guzmán, P., Carro, M., Warren, D.S.: Swapping Evaluation: A Memory-Scalable Solution for Answer-On-Demand Tabling. TPLP 10 (4-6), 401–416 (2010)
4. Chico de Guzmán, P., Carro, M., Hermenegildo, M.V., Stuckey, P.: A general implementation framework for tabled CLP. In: Schrijvers, T., Thiemann, P. (eds.) FLOPS 2012. LNCS, vol. 7294, pp. 104–119. Springer, Heidelberg (2012)
5. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)
6. Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.: XMC: A Logic-Programming-Based Verification Toolset. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 576–580. Springer, Heidelberg (2000)
7. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems 20(3), 586–634 (1998)
8. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. TPLP 12(1-2), 157–187 (2012)
9. Rocha, R.: Handling Incomplete and Complete Tables in Tabled Logic Programs. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 427–428. Springer, Heidelberg (2006)
10. Sagonas, K.F., Stuckey, P.J.: Just Enough Tabling. In: PPDP 2004, pp. 78–89. ACM (August 2004)
11. Guo, H.F., Gupta, G.: Cuts in Tabled Logic Programming. In: Demoen, B. (ed.) CICLOPS 2002, pp. 62–73 (July 2002)
12. Guo, H.F., Gupta, G.: Simplifying Dynamic Programming via Mode-directed Tabling. Softw. Pract. Exper. 38(1), 75–94 (2008)
13. Castro, L.F., Warren, D.S.: Approximate Pruning in Tabled Logic Programming. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 69–83. Springer, Heidelberg (2003)
14. Guo, H.-F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 181–196. Springer, Heidelberg (2001)

A Portable and Efficient Implementation of Coinductive Logic Programming

Paulo Moura

CRACS, INESC TEC (formerly INESC Porto), Portugal
pmoura@inescporto.pt

Abstract. We describe the portable and efficient implementation of coinductive logic programming found in Logtalk, discussing its features and limitations. As Logtalk uses as a back-end compiler a compatible Prolog system, we also discuss the status of key Prolog features for an efficient and usable implementation of coinduction.

Keywords: logic programming, coinduction, objects, implementation, portability.

1 Introduction

Coinductive logic programming complements classical inductive logic programming by allowing writing of programs that reason about infinite rational entities such as cyclic terms and ω-automata. Areas of application include modeling and verification of real-time systems [1,2] and lazy evaluation [3].

This paper describes the current implementation of coinductive logic programming found in Logtalk, discussing its features and limitations.[1] As Logtalk uses as a back-end compiler a compatible Prolog system, we also discuss the status of key Prolog features for an efficient and usable implementation of coinduction. We assume that the reader is familiar with the theoretical work in coinduction (see e.g. [4,5]). Therefore, this paper is written from a practical, technical point-of-view.

The main motivation for implementing support for coinductive logic programming in Logtalk is to make it the preferred tool for solving problems that require coinductive reasoning. This is an ambitious and long term goal, but we believe that the core features of Logtalk, including its code encapsulation and code reuse mechanisms, provide a strong framework for solving complex problems where coinduction is one of the solution components. In addition, the inherent requirements on back-end Prolog compiler native features, for example, on support for rational terms, tabling, and constraints, hopefully help drive future enhancements to Prolog implementations that will ultimately benefit the logic programming community at large.

[1] The first Logtalk implementation of coinductive logic programming was introduced in version 2.41.0, released on September 15, 2010.

The remainder of the paper is organized as follows. Section 2 provides an overview of Logtalk. Section 3 describes the coinductive predicate directives provided by Logtalk. Section 4 describes the hook predicates that support user-customization of the coinductive proof algorithm. Section 5 describes in detail our implementation of coinduction, discussing its features and limitations. Section 6 presents some examples of coinductive predicates. Section 7 shows Logtalk built-in support for debugging coinductive predicates. Section 8 compares our implementation with related work. Section 9 concludes and outlines future work.

2 Logtalk in a Nutshell

Logtalk [6,7] is an open source object-oriented logic programming language that can use most Prolog implementations as back-end compilers. Logtalk focuses on code encapsulation and code reuse features, providing a versatile alternative to Prolog module systems. As a multi-paradigm language, Logtalk supports classes, prototypes, parametric objects, categories (fine-grained units of code reuse), separation between interface and implementation using protocols, event-driven programming, and high-level multi-threading programming. Logtalk uses *object* as a generic term: an object can play the role of, e.g., an instance, a class, or a prototype. The relations between objects, protocols, and categories define different *patterns of code reuse*. Logtalk entities can be static, defined in source files, or dynamic, created at runtime. Computations are performed by sending *messages* (corresponding to predicates) to objects. Logtalk enforces predicate encapsulation (predicates can be declared public, protected, or private) and features a clear distinction between predicate declaration and predicate definition (using a closed-world assumption when a predicate is declared but not defined). Logtalk is developed with a strong emphasis on portability and reliability. It is used worldwide in academic and commercial projects. Its distribution includes extensive documentation, numerous examples, a library, and basic development tools (for debugging, unit testing, and documenting).

3 Coinductive Predicate Directives

Logtalk requires coinductive predicates to be explicitly declared, as the predicate clauses must be compiled with support for checking coinductive success and for keeping a stack of coinductive hypotheses. When constructing a proof for a coinductive predicate goal, the coinductive hypotheses are the ancestor goals for the same coinductive predicate. Coinductive success is achieved when the current goal unifies with a coinductive hypothesis. We can have multiple proofs, and thus possibly multiple solutions, when the current goal unifies with more than one coinductive hypothesis.

Coinductive predicates are declared using the `coinductive/1` predicate directive. The argument of this directive can be a predicate indicator when all the predicate arguments are relevant for coinductive success. As an example, consider Listing 1.1.

Listing 1.1. Infinite lists with a repeating pattern of binary digits

```
:- object(binary).

    :- public(p/1).
    :- coinductive(p/1).
    p([0| T]) :- p(T).
    p([1| T]) :- p(T).

:- end_object.
```

When only some arguments should be considered when testing for coinductive success, the directive argument must be a predicate template. In this case, coinductive predicate arguments are represented by the atom `'+'`, while arguments that should be disregarded are represented by the atom `'-'`. Listing 1.2 illustrates an example. In this case, we want to find the cyclic paths in a graph whose length (of the repeating pattern) is bound by a given value.

Listing 1.2. Length-limited cyclic paths in a graph

```
:- object(cyclic_paths).

    :- public(path/3).
    path(From, Path, MaxLength) :-
        path(From, Path, 0, MaxLength).

    :- private(path/4).
    :- coinductive(path(+, +, -, -)).
    path(From, [From| Path], Length, MaxLength) :-
        arc(From, Next),
        Length < MaxLength,
        Length1 is Length + 1,
        path(Next, Path, Length1, MaxLength).

    arc(a, b).
    arc(b, c).
    arc(c, a).  arc(c, d).
    arc(d, a).

:- end_object.
```

In this case, coinductive success depends only on the first two arguments of the `path/4` auxiliary predicate. The remaining two arguments are only used to limit the solutions found and are ignored when checking for coinductive success.

This representation of relevant arguments is the same representation used in predicate tabling directives in systems such as B-Prolog, where it is possible to indicate which arguments should be considered for variant checking, allowing selective tabling of answers. The use of a common representation for declaring relevant predicate arguments for coinductive success and for variant checking when tabling predicate answers may provide, however, benefits other than language consistency. Intuitively, we expect that the arguments that are relevant for coinductive success are the same that are relevant for variant checking. This would mean that the `coinductive/1` predicate directive would make writing tabling directives for the same predicates redundant, simplifying programming.

4 Coinductive Success Hook Predicates

Hook predicates are a common solution for user customization of system-implemented algorithms and mechanisms. They may also be used for debugging, by allowing tracing of the hooked steps.

In the specific case of coinduction, a generic hook predicate, `essence_hook/2`, is supported by the DRA meta-interpreter[8]. In this case, the hook predicate is primarily intended to allow the specification of the relevant predicate arguments for coinductive success. But, according to the documentation, it may also be used for defining an alternative to unification when checking for coinductive success and for calling user code when tabling an answer or using a table answer (as discussed in the previous section, Logtalk uses an extended `coinductive/1` directive for specifying the relevant predicate arguments for coinductive success).

More recently, [9] proposes two hook predicates, `finally/1-2`, whose usefulness is demonstrated with several examples. The author shows how these hooks allow implementation solutions for applications which otherwise would require tabling support. The current Logtalk development version[2] implements these two hook predicates but under the `coinductive_success_hook/1-2` alternative names. The Logtalk compiler optimizes the calls to these hook predicates and ensures zero overhead for the coinductive predicates that do not use them. These hook predicates are called in the case of coinductive success. The first argument is the term resulting from the unification of the current goal with a coinductive hypothesis. The second argument, when present, is the used coinductive hypothesis. Listing 1.3 shows an example, adapted to Logtalk from [9], of testing for and enumerating the elements of a *rational* list. An alternative tabling-based definition is illustrated in Listing 1.7.

Listing 1.3. Testing and enumerating elements of a rational list

```
:- object(lists).

    % Are there "occurrences" of arg1 in arg2?
    :- public(member/2).
    :- coinductive(member/2).
    member(X, [X| _]).
    member(X, [_| T]) :-
        member(X, T).
```

[2] Publicly available from `https://github.com/LogtalkDotOrg/logtalk3`

```
    % Are there infinitely many "occurrences" of arg1 in arg2?
    :- public(comember/2).
    :- coinductive(comember/2).
    comember(X, [_| T]) :-
        comember(X, T).

    coinductive_success_hook(member(_, _)) :-
        fail.
    coinductive_success_hook(comember(X, L)) :-
        member(X, L).

:- end_object.
```

The idea behind this solution is that the definition of the `comember/2` traverses the list until it finds the repeating pattern (achieving coinductive success at that point), thus skipping any existing prefix. When that happens, the `member/2` predicate enumerates the elements in the repeating pattern, thanks to the second clause for the `coinductive_success_hook/1` predicate. The first clause of the hook predicate ensures termination of a call to the `member/2` predicate when coinductive success is achieved.

5 Implementation

A coinductive predicate is compiled by adding a *preflight predicate* that checks for coinductive success and, if not yet achieved, pushes the current goal to the stack of coinductive hypotheses (i.e., the ancestor goals for the coinductive predicate query). This preflight predicate calls the coinductive predicate defined by the programmer. The user clauses are modified by replacing the recursive call to the coinductive predicate by a call to the preflight predicate. The per-object table of defined predicates ensures that a message corresponding to the coinductive predicate is translated to a call to the preflight predicate.

The stack of coinductive hypotheses is represented using a list and passed between predicate calls using a hidden extra argument that is used for representing the execution context. This extra argument is added by the Logtalk compiler to the compiled form of all predicates.[3] An alternative implementation of the coinductive hypotheses stack would be to use the destructive assignment built-in predicates that are found on some Prolog compilers. But these predicates are not standard and our goal is a portable implementation.

Checking for coinductive success is performed by attempting to unify the current goal with an elements of the coinductive hypotheses stack. This unification may succeed, on backtracking, for more than one hypothesis, thus leading to

[3] Logtalk uses an extra predicate argument for passing execution context information, which includes the *sender* of a message and the object that received the message (*self*). This allows a simple implementation of the stack of coinductive hypotheses as just an additional argument of the execution context term.

multiple solutions. On the other hand, the current goal is only pushed to the stack of coinductive hypotheses if it does not unify with any of its elements. This semantics is efficiently implemented using the *soft-cut* control construct found on several Prolog compilers, including all of those that provide the necessary minimal support for rational terms.[4]

The following example of the compilation of the coinductive predicate `p/1` in Listing 1.1 illustrates our current implementation (with all non-relevant details, including the internal names of the coinductive and preflight predicates, abstracted for clarity of presentation):

Listing 1.4. Compiled code for a coinductive predicate `p/1`

```
p_1_coinduction_preflight(A, Stack) :-
    (   member(p(A), Stack) *->
        true
    ;   p(A, [p(A)| Stack])
    ).

p([0| A], Stack) :-
    p_1_coinduction_preflight(A, Stack).
p([1| A], Stack) :-
    p_1_coinduction_preflight(A, Stack).
```

In the code above, the predicate `member/2` has its traditional inductive definition and the `(*->)/2` operator denotes the soft-cut control construct, as found on several Prolog compilers such as ECLiPSe, GNU Prolog, SWI-Prolog, and YAP.[5]

When the `coinductive_success_hook/1` or the `coinductive_success_hook/2` hook predicate are defined for a coinductive predicate, they are called in the place of the goal `true/0` in the code in Listing 1.4 (the Logtalk compiler looks first for a user definition of the arity two version of the hook predicate).

5.1 Implementation Limitations

In the current Logtalk implementation, the stratification of programs mixing non-coinductive predicates and coinductive predicates is neither checked nor enforced. Thus, ensuring stratification is a responsibility left to the programmer.

A second, more fundamental limitation is partially a consequence of the lack of native Prolog support for tabling of rational terms (see Section 5.3). The practical consequence is that, while coinductive predicates can *recognize* any valid solution, they can only *generate* a (finite) subset of all possible solutions.

[4] In this paper, we use the usual definition of *rational term*: an infinite term with a finite representation.

[5] Some other Prolog compilers such as SICStus Prolog use a built-in meta-predicate, `if/3`, for implementing a soft-cut. Logtalk uses either the `(*->)/2` control construct or the `if/3` built-in meta-predicate depending on the used back-end Prolog compiler.

For example, using the coinductive predicate `p/1` in Listing 1.1, we get the results illustrated in Listing 1.5.[6]

Listing 1.5. Solutions generated for the coinductive predicate `p/1` in Listing 1.1

```
?- binary::p(X).
X = [0|X] ;
X = [1|X] ;
false.

?- L = [0,1,0| L], binary::p(L).
L = [0, 1, 0|L] ;
false.
```

We describe the finite set of generated solutions as the set of *basic cycles*, where a basic cycle is a solution that cannot be expressed as a combination of other solutions. Ideally, any possible solution could be generated from a *combination* of these basic cycles. But we do not have yet a formal proof and our intuition can be wrong. With tabling support available, we could use an alternative compilation scheme where the current goal would be added to the stack of coinductive hypotheses, independently of the current goal unifying with any of the existing coinductive hypotheses. Without tabling, and for the example in Listing 1.1, this alternative compilation scheme repeatedly generates, as expected, and as long as memory is available, the first solution, as illustrated in Listing 1.6. With a suitable tabling implementation, we would not get stuck repeating the same solution, but we could still get an infinite number of solutions. As an alternative, a breadth-first inference mechanism can also avoid repeatedly generating the same solution. In fact, this approach is used in one of the variations of the U.T.Dallas Prolog meta-interpreter for coinductive predicates. But a solution where we generate the finite set of basic cycles and use it to construct an *expression* representing all possible combinations of these basic cycles would be preferable as this expression could then be used to both generate and test solutions as necessary.

Listing 1.6. Solutions generated for the coinductive predicate `p/1` in Listing 1.1 using the alternative compilation scheme

```
?- binary::p(X).
X = [0|X] ;
X = [0|_S1], % where
    _S1 = [0|_S1] ;
X = [0, 0|X] ;
X = [0, 0|_S1], % where
    _S1 = [0|_S1] ;
...
```

[6] Using SWI-Prolog as the Logtalk back-end compiler.

5.2 Implementation Portability

The current coinduction implementation supports a subset of the Logtalk compatible back-end Prolog compilers. Namely, ECLiPSe, SICStus Prolog, SWI-Prolog, and YAP. The two main Prolog native features necessary for our implementation are (1) a soft-cut control construct or built-in predicate[7] and (2) minimal support for rational terms. The soft-cut control construct is already implemented or is being implemented on most Prolog compilers. The most problematic feature is the the support for rational terms, as we discuss next.

5.3 Rational Terms Support

Although an implementation of coinductive logic programming must be able to create, unify, and print bindings with rational terms, there is very limited standard support for this kind of terms. The latest official revision of the ISO Prolog Core standard [10] added an `acyclic_term/1` built-in predicate but does not specify a comprehensive set of *operations* on rational terms that should be supported. In addition, for a long time, rational terms were regarded more as a problem than as a feature in Prolog compilers. Thus, the supported operations on rational terms depend on the Prolog compiler. Fortunately, implementing coinduction requires only three basic operations: (1) creation of rational terms, (2) unification of rational terms, and (3) a suitable printing of rational terms, such that bindings resulting from queries to coinductive predicates can be non-ambiguously interpreted. Creating and unifying rational terms are supported by all compatible back-end Prolog compilers. But non-ambiguous printing of rational terms is a problem for most compilers. To illustrate the problem, consider the `p/1` coinductive predicate in Listing 1.1 and the query `p(X)`. Our implementation provides two solutions for this query, the rational terms `X = [0|X]` and `X = [1|X]`. The solutions as printed by ECLiPSe, SICStus Prolog, SWI-Prolog, and YAP are presented in Table 1.

Table 1. Printing of rational terms bindings

Prolog compiler	**First solution**	**Second solution**
ECLiPSe 6.1.115	`X = [0, 0, 0, 0, ...]`	`X = [1, 1, 1, 1, ...]`
SICStus Prolog 4.0.4	`X = [0, 0, 0, 0, ...]`	`X = [1, 1, 1, 1, ...]`
SWI-Prolog 6.1.11	`X = [0|X]`	`X = [1|X]`
YAP 6.3.2	`X = [0|**]`	`X = [1|**]`

The only reason we do not get into trouble when using ECLiPSe and SICStus Prolog is that both limit, by default, the maximum length of a list when printing

[7] Although it is possible to implement the preflight predicate without using a soft-cut, the resulting code would provide poor performance as it would require, in the worst case, traversing the list that implements the stack of coinductive hypotheses twice.

terms.[8] YAP prints an ambiguous mark, `**`, to alert the user that is printing a rational term. Only SWI-Prolog provides a non-ambiguous printing of rational terms bindings.

5.4 Tabling of Rational Terms

The set of coinductive problems that can be tackled by the current implementation is limited by the lack of a compatible back-end Prolog compiler that natively supports tabling of rational terms. A simple example where tabling is required is in the following alternative definition of the `comember/2` predicate. This predicate succeeds when an element occurs an infinite number of times in a list. It can be defined as illustrated in Listing 1.7.

Listing 1.7. Definition of the coinductive predicate `comember/2`

```
:- coinductive(comember/2).
comember(X, L) :-
    drop(X, L, L1),
    comember(X, L1).

:- table(drop/3).
drop(H, [H| T], T).
drop(H, [_| T], T1) :-
    drop(H, T, T1).
```

The auxiliary predicate `drop/3` is used to drop elements from the input list non-deterministically. But, without tabling support for rational terms, the call to this predicate in the definition of the `comember/2` will unify the first element and will limit the coinductive predicate to return that solution repeatedly (on backtracking) without ever moving to the next solution.

Although it is always possible to implement a high-level tabling solution, the relatively poor performance of such solution makes it preferable to work with Prolog implementers that already provide a native tabling system in extending it to support rational terms. The current alternative, as illustrated in Section 4, can provide a good alternative. However, more experience is necessary for meaningfully compare the programmer effort of writing the necessary hook predicate definitions versus writing tabling-based solutions for the same problems. In addition, the programmer must be aware that tabling-based solutions can feature better complexity properties by avoiding recomputing solutions.

5.5 Coroutining and CLP(R) Libraries

Some of the recent research on coinduction focuses on model checking and timed automata (see e.g. [11,2,12]). The implementation of solutions for these classes of

[8] ECLiPSe uses, by default, a `depth(20)` output option. SICStus Prolog uses, by default, a `max_depth(10)` output option.

problems require the use of coroutining and CLP(R) libraries. All the back-end Prolog compilers we support for coinduction provide built-in coroutining primitives and these constraint libraries, although the ECLiPSe versions differs in its interface from those found on SICStus Prolog, SWI-Prolog, and YAP. Logtalk can account for the differences using its conditional compilation directives. Not an ideal solution, however, as it still results in code duplication. But there are two other, more significant, potential issues: lack of active maintenance of some of these libraries and semantic differences between the different implementations of coroutining and constraint libraries. In fact, there is currently no standardization effort for constraint programming in Prolog, despite the area being widely recognized as fundamental for the practical success of logic programming.

6 Examples

The current Logtalk distribution includes sixteen coinduction examples, most of them adapted from publications on coinductive logic programming or originating from discussions with researchers in this area. The examples are complemented by unit tests, thus providing a handy solution for testing our implementation across compatible back-end Prolog compilers. In this section, we present and briefly discuss some of the most interesting examples, mainly to familiarize the reader on how to define coinductive predicates. The example queries output are produced using Logtalk with SWI-Prolog as the back-end compiler.

6.1 A Tangle of Coinductive Predicates

Our first example (Listing 1.8) illustrates a coinductive predicate with two starting points and no common solution prefix. This example was originally written by Feliks Kluźniak in the context of a discussion on how to combine coinductive predicate solutions to construct other valid solutions.

Listing 1.8. A coinductive predicate with two starting points and no common solution prefix, wrapped in a `tangle` object

```
:- object(tangle).

    :- public(p/1).
    :- coinductive(p/1).
    p([a| X]) :- q(X).
    p([c| X]) :- r(X).

    :- coinductive(q/1).
    q([b| X]) :- p(X).

    :- coinductive(r/1).
    r([d| X]) :- p(X).

:- end_object.
```

Listing 1.9 shows two queries for the `tangle::p/1` predicate. The first query works as a solution generator, while the second query tests a specific solution.

Listing 1.9. Sample queries for the `tangle::p/1` coinductive predicate

```
?- tangle::p(X).
X = [a, b|X] ;
X = [c, d|X] ;
false.

?- L = [a, b, c, d| L], tangle::p(L).
L = [a, b, c, d|L] ;
false.

?- L = [a, c| L], tangle::p(L).
false.
```

6.2 An Omega-Automaton

Our third example (Listing 1.10) is adapted from [3] and illustrates an ω-automaton, i.e. an automaton that accepts infinite strings. The commented out code shows how we can go from an automaton recognizing finite strings to an ω-automaton by simply dropping the base case in the recursive definition.

Listing 1.10. A omega-automaton

```
:- object(automaton).

    :- public(automaton/2).
    :- coinductive(automaton/2).

    automaton(State, [Input| Inputs]) :-
        trans(State, Input, NewState),
        automaton(NewState, Inputs).
%   automaton(State, []) :- % we drop the base case in order
%       final(State).       % to get an omega-automaton

    trans(s0, a, s1).
    trans(s1, b, s2).
    trans(s2, c, s3).
    trans(s2, e, s0).
    trans(s3, d, s0).

    final(s2).

:- end_object.
```

Listing 1.11 shows generating and testing queries for the `automaton::automaton/2` coinductive predicate.

Listing 1.11. Sample queries for the `automaton::automaton/2` predicate

```
?- automaton::automaton(s0, X).
X = [a, b, c, d|X] ;
X = [a, b, e|X] ;
false.

?- L = [a, b, c, d, a, b, e| L], automaton::automaton(s0, L).
L = [a, b, c, d, a, b, e|L] ;
false.

?- L = [a, b, e, c, d| L], automaton::automaton(s0, L).
false.
```

6.3 A Sieve of Eratosthenes Coinductive Implementation

The second example (Listing 1.12) presents our coinductive implementation of the Sieve of Eratosthenes. An alternative solution, based on coroutining, is sketched in [3].

Listing 1.12. A Sieve of Eratosthenes coinductive implementation

```
:- object(sieve).

    :- public(primes/2).
    % computes a coinductive list with all the
    % primes in the 2..N interval
    primes(N, Primes) :-
        generate_infinite_list(N, List),
        sieve(List, Primes).

    % generate a coinductive list with a 2..N
    % repeating pattern
    generate_infinite_list(N, List) :-
        sequence(2, N, List, List).

    sequence(Sup, Sup, [Sup| List], List) :-
        !.
    sequence(Inf, Sup, [Inf| List], Tail) :-
        Next is Inf + 1,
        sequence(Next, Sup, List, Tail).

    :- coinductive(sieve/2).
    sieve([H| T], [H| R]) :-
        filter(H, T, F),
        sieve(F, R).


    :- coinductive(filter/3).
    filter(H, [K| T], L) :-
        (   K > H, K mod H =:= 0 ->
            % throw away the multiple we found
            L = T1
        ;   % we must not throw away the integer used for
            % filtering in order to return a filtered
            % coinductive list
            L = [K| T1]
        ),
        filter(H, T, T1).

:- end_object.
```

Listing 1.13 illustrates how to use our `sieve::primes/2` coinductive predicate to enumerate all the prime numbers in the `[1..20]` interval.

Listing 1.13. Enumerating prime numbers using coinduction

```
?- sieve::primes(20, P).
P = [2, 3|_S1], % where
    _S1 = [5, 7, 11, 13, 17, 19, 2, 3|_S1] .
```

7 Debugging Coinductive Predicates

As most extensions to existing logic programming languages, the practical use of coinduction depends not only on robust implementations with good performance but also on development tools support, in particular for debugging. Logtalk provides specific support for debugging coinductive predicates by allowing (1) tracing of coinductive success checks, (2) tracing of pushing the current goal to the stack of coinductive hypotheses, and (3) printing of the stack of coinductive hypotheses at any time. Operations (1) and (2) are collectively described as a *coinduction preflight* step, which takes place at every coinductive predicate call before proceeding to the clauses defined by the programmer (as detailed in Section 5). The example in Listing 1.14 shows a debugging section (with internal variable names renamed for clarity and using SWI-Prolog as the Logtalk back-end compiler).

Listing 1.14. Debugging a coinductive predicate call

```
?- binary::p(X).
  Call: (1) binary::p(X) ?
  Rule: p_1_coinduction_preflight(X) ?
  Call: (2) check_coinductive_success(p(X),[]) ?
  Fail: (2) check_coinductive_success(p(X),[]) ?
  Call: (3) push_coinductive_hypothesis(p(X),[],S) ?
  Exit: (3) push_coinductive_hypothesis(p(X),[],[p(X)]) ?
  Call: (4) p(X) ?
  Rule: (clause #1) p([0|L]) ?
  Call: (5) p_1_coinduction_preflight(L) ? x
  Sender:           user
  This:             binary
  Self:             binary
  Meta-call context: []
  Coinduction stack: [p([0|L])]
  Call: (5) p_1_coinduction_preflight(L) ?
  Rule: p_1_coinduction_preflight(L) ?
  Call: (6) check_coinductive_success(p(L),[p([0|L])]) ?
  Exit: (6) @(check_coinductive_success(p(S_1),[p(S_1)]),
                                            [S_1=[0|S_1]]) ?
  Call: (7) true ?
  Exit: (7) true ?
  Exit: (5) @(p_1_coinduction_preflight(S_1),[S_1=[0|S_1]]) ?
  Exit: (4) @(p(S_1),[S_1=[0|S_1]]) ?
  Exit: (1) @(binary::p(S_1),[S_1=[0|S_1]]) ?
X = [0|X] ;
...
```

8 Related Work

The U.T.Dallas research group on coinduction makes available a Prolog meta-interpreter, implemented by Feliks Kluźniak in 2009, that supports both tabling and coinduction [8]. The meta-interpreter distribution includes example applications for the model checker. Although the meta-interpreter suffers from slower performance when compared with the Logtalk implementation, the high-level implementation of tabling allows it to solve a wider class of problems, without being dependent on native Prolog tabling support. For problems that do not require tabling, the U.T.Dallas implementation provides a simple program transformer that adds an extra argument (with the stack) to coinductive predicates, thus enabling them to be executed without the overhead of interpretation.

Two Prolog compilers, SWI-Prolog and YAP, include limited support for coinduction, implemented by a library module. The YAP implementation takes advantage of non-portable primitives for destructive assignment for representing the coinduction stack when constructing a proof for a coinductive predicate. The SWI-Prolog implementation uses proprietary hook predicates to access a goal and its parent goal during a proof. Although these choices render the implementations non-portable, they also make them potentially more efficient than a portable implementation such as the one found in Logtalk. Both the SWI-Prolog and YAP implementations only support the most simple form of the coinductive directive where only a predicate indicator can be specified. As in the current Logtalk implementation, stratification of programs mixing non-coinductive predicates and coinductive predicates is neither checked nor enforced.

A set of Prolog meta-interpreters for coinductive logic programming are presented and discussed in [9], together with several illustrating examples. Although these meta-interpreters are best viewed as a proof-of-concept (giving the inherent performance penalty of meta-interpretation), they clearly illustrate several problems and conceptual solutions when implementing coinductive logic programming. In particular, the proposed hook predicates provide a practical and strong alternative to tabling for some problems. These hook predicates are efficiently implemented in the latest Logtalk development versions.

9 Conclusions and Future Work

Logtalk provides a widely available and portable implementation of coinductive logic programming. It features basic coinductive debugging support and includes several examples that are complemented by unit tests. It can be easily used for demoing the basic ideas of coinductive logic programming in the classroom and for solving actual problems. Its implementation avoids meta-interpretation by compiling both coinductive predicate definitions and any used hook predicates, thus providing good performance for coinduction applications that do not require tabling support.

The current implementation is designed with the intuition is that it can generate, by backtracking, all *basic cycles*, whose combinations account for all possible solutions. If our intuition is correct, it should be possible to derive an *expression* that represents that combination and that can be used for checking or generating any solution. Assuming that deriving such an expression can be soundly accomplished in practice and for any problem, this would provide a potential alternative to all current implementations, which all suffer from the fact that an infinite set of solutions cannot be enumerated in a finite time. Thus, the problem of how to discover all basic cycles and how to combine them in an expression appears to be the most interesting open problems and thus a promising line for future work.

Our plans for better coinduction support in Logtalk, while maintaining or improving portability, are partially dependent on the evolution of the compatible Prolog systems. There are two main issues. First, printing of rational terms, which is used when printing bindings for solutions to coinductive queries, only works acceptably on SWI-Prolog. For all the other supported back-end Prolog compilers, the bindings printed are often ambiguous. Second, tabling of rational terms. This will enable Logtalk to tackle problems that cannot currently be solved or that can be solved but with non-practical time/space complexity. We plan to work closely with Prolog implementers on solving both issues.

We are also following progress on the theoretical aspects of coinduction, specially when combined with constraint programming, and hope to be able to implement new, proven, ideas when feasible and in a timely manner.

Acknowledgements. We thank Gopal Gupta, Feliks Kluźniak, Neda Saeedloei, Brian DeVries, and Kyle Marple for helpful discussions on coinduction during a sabbatical visit to U.T.Dallas and for most of the coinduction examples that are currently adapted and distributed with Logtalk. We thank Davide Ancona for feedback on hook predicates and the corresponding examples. We also thank Jan Wielemaker and Vitor Santos Costa for feedback on using destructive assignment primitives for representing coinduction stacks and on representing and printing rational terms. A preliminary version of this work was presented on the 1st Workshop on Coinductive Logic Programming (CO-LP 2012). This revised and extended version includes a new section on coinductive success hook predicates and a revised discussion on tabling support. This work is partially supported by the LEAP project (PTDC/EIA-CCO/112158/2009), the ERDF/COMPETE Program and by FCT project FCOMP-01-0124-FEDER-022701.

References

1. Saeedloei, N., Gupta, G.: A logic-based modeling and verification of CPS. SIGBED Rev. 8, 31–34 (2011)
2. Saeedloei, N.: Modeling and Verification of Real-Time and Cyber-Physical Systems. PhD thesis, University of Texas at Dallas, Richardson, Texas (2011)
3. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and Its Applications. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 27–44. Springer, Heidelberg (2007)
4. Gupta, G., Saeedloei, N., DeVries, B., Min, R., Marple, K., Kluźniak, F.: Infinite Computation, Co-induction and Computational Logic. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 40–54. Springer, Heidelberg (2011)
5. Simon, L.: Coinductive Logic Programming. PhD thesis, University of Texas at Dallas, Richardson, Texas (2006)
6. Moura, P.: From Plain Prolog to Logtalk Objects: Effective Code Encapsulation and Reuse (Invited Talk). In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 23–23. Springer, Heidelberg (2009)
7. Moura, P.: Logtalk - Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
8. Kluźniak, F.: Metainterpreter supporting tabling (DRA) and coinduction with applications to LTL model checking, `http://www.utdallas.edu/~gupta/meta.html`
9. Ancona, D.: Regular corecursion in Prolog. In: Ossowski, S., Lecca, P. (eds.) SAC, pp. 1897–1902. ACM (2012)
10. ISO/IEC: International Standard ISO/IEC 13211-1 Information Technology — Programming Languages — Prolog — Part I: General core, Corrigenda 2. ISO/IEC (2012)
11. Saeedloei, N., Gupta, G.: Coinductive Constraint Logic Programming. In: Schrijvers, T., Thiemann, P. (eds.) FLOPS 2012. LNCS, vol. 7294, pp. 243–259. Springer, Heidelberg (2012)
12. Saeedloei, N., Gupta, G.: Verifying complex continuous real-time systems with coinductive CLP(R). In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 536–548. Springer, Heidelberg (2010)

Formalizing a Broader Recursion Coverage in SQL

Gabriel Aranda[1], Susana Nieva[1],
Fernando Sáenz-Pérez[2], and Jaime Sánchez-Hernández[1,⋆]

[1] Dept. Sistemas Informáticos y Computación, UCM, Spain
[2] Dept. Ingeniería del Software e Inteligencia Artificial, UCM, Spain
garanda@fdi.ucm.es, {nieva,fernan,jaime}@sip.ucm.es

Abstract. SQL is the *de facto* standard language for relational databases and has evolved by introducing new resources and expressive capabilities, such as recursive definitions in queries and views. Recursion was included in the SQL-99 standard, but this approach is limited as only linear recursion is allowed, mutual recursion is not supported, and negation cannot be combined with recursion. In this work, we propose a new approach, called R-SQL, aimed to overcome these limitations and others, allowing in particular cycles in recursive definitions of graphs and mutually recursive relation definitions. In order to combine recursion and negation, we import ideas from the deductive database field, such as stratified negation, based on the definition of a dependency graph between relations involved in the database. We develop a formal framework using a stratified fixpoint semantics and introduce a proof-of-concept implementation.

Keywords: Databases, SQL, Recursion, Fixpoint Semantics.

1 Introduction

Codd's famous paper on relational model [2] sowed the seeds for current relational database management systems (RDBMS's), such as DB2, Oracle, MySQL, SQL Server and others. Formal query languages were proposed for the relational model: Relational algebra (RA) and relational calculus, which are syntactically different but semantically equivalent w.r.t. safe formulas [16]. Such RDBMS's rather rely on the SQL query language (current standard SQL:2008 [7]) that departs from the relational model and goes beyond. Its acknowledged success builds upon an elegant and yet simple formulation of a data model with relations which can be queried with a language including some basic RA-operators, which are all about relations. Original operators became a limitation for practical applications of the model, and others emerged to fill some gaps, including, for instance, aggregate operators for, e.g., computing running sums and averages.

⋆ This work has been partially supported by the Spanish projects TIN2008-06622-C03-01 (FAST-STAMP), S2009/TIC-1465 (PROMETIDOS), and GPD-UCM-A-910502.

Other additions include representing absent or unknown information, which delivered the introduction of null values and outer join operators ranging over such values. Also, duplicates were introduced to account for bags (multisets) instead of sets. Finally, we mention the inclusion of recursion (Starburst [10] was the first non-commercial RDBMS to implement this whereas IBM DB2 was the first commercial one), a powerful feature to cope with queries that must be otherwise solved by the intermixing with a host language. However, as pointed out by many (see, e.g., [9],[13]), the relational model has several limitations. Thus, such current RDBMS's include that extended "relational" model, which is far from the original one and it is even heavily criticized [3] because of nulls and duplicates.

In this work, we focus on the inclusion of recursion in SQL as current RDBMS's lack both a formal support and suffer a narrow coverage of recursion. Regarding formalization, an extension of the RA is presented in [1], with a looping construct and assignment in order to deal with the integration of recursion and negation. [5] is the source of the original SQL-99 proposal for recursion, which is based on the research in the areas of logic programming and deductive databases [16], as explained in [4]. Another example of an approach built on an extension of RA with a fixpoint construct is in [6]. However, as far as we know, these formalizations do not lead to concrete implementations, while our proposal provides an operational mechanism allowing a straightforward implementation.

Regarding recursion coverage, there are several main drawbacks in current implementations of recursion: Linearity is required, so that relation definitions with calls to more than one recursive relation are not allowed. Some other features are not supported: Mutual recursion, and query solving involving an EXCEPT clause. In general, termination is manually controlled by limiting the number of iterations instead of detecting that there are no further opportunities to develop new tuples.

Here, we propose R-SQL, a subset of the SQL standard to cope with recursive definitions which are not restricted as current RDBMS's do, and also allowing neater formulations by allowing concise relation definitions (much following the assignment RA-operator) and avoiding extensive writings (cf. Section 2). For this language, first we develop a novel formalization based on stratified interpretations and a fixpoint operator to support theoretical results (cf. Section 3). And, second, we propose a proof-of-concept implementation which takes a set of database relation (in general, recursive) definitions and computes their meanings (cf. Section 4). This implementation uses the underlying host SQL system and Python to compute the outcome, and can be easily adapted to be integrated as a part of any state-of-the-art RDBMS. Section 5 concludes and presents some further work.

2 Introducing R-SQL

In this section, we present the language R-SQL by using a minimal syntax that allows to capture the core expressiveness of standard SQL. Namely, we consider basic SQL constructs to cover relational algebra. Nevertheless, this language is conceived to be able to be extended in order to incorporate other usual features. R-SQL is focused on the incorporation of recursive relation definitions. The idea is simple and effective: A relation is defined with an assignment operation as a named query (view) that can contain a self reference, i.e., a relation `R` can be defined as `R sch :=` SELECT ... FROM ... `R` ..., where `sch` is the relation schema. Next, we introduce the formal grammar of this language, then we show by means of examples the benefits of R-SQL w.r.t. current RDBMS systems.

2.1 Syntax of R-SQL

The formal syntax of R-SQL is defined by the grammar in Figure 1. In this grammar, productions start with lowercase letters whereas terminals start with uppercase (SQL terminal symbols use small caps). Optional statements are delimited by square brackets and alternative sentences are separated by pipes. The grammar defines the following syntactic categories:

- A database `sql_db` is a (non-empty) sequence of relation definitions separated by semicolons (";"). A relation definition assigns a select statement to the relation, that is identified by its name `R` and its schema.
- A schema `sch` is a tuple of attribute names with their corresponding types.
- A select statement `sel_stm` is defined in the usual way. The clauses FROM and WHERE are optional. We also allow UNION and EXCEPT, but notice that the syntax for EXCEPT allows only a relation name instead of a select statement

```
sql_db  ::= R sch := sel_stm;
            ...
            R sch := sel_stm;

sch     ::= (A T,...,A T)

sel_stm ::= SELECT exp,...,exp [ FROM R,...,R [ WHERE wcond ] ]
            | sel_stm UNION sel_stm
            | sel_stm EXCEPT R

exp     ::= C | R.A | exp opm exp | - exp

wcond   ::= TRUE | FALSE | exp opc exp | NOT (wcond)
            | wcond [ AND | OR ] wcond

opm     ::= + | - | / | *

opc     ::= = | <> | < | > | >= | <=
```

`R` stands for relation names, `A` for attribute names, `T` for standard SQL types (as INTEGER, FLOAT, VARCHAR(N)), and `C` for constants belonging to a valid SQL type.

Fig. 1. A Grammar for the R-SQL Language

as usual in SQL. This is done in order to keep simple the syntax and does not imply expressivity losses, because a relation name can be identified with the select statement that defines it.
– An expression `exp` can be either a constant value `C`, an attribute of a relation (denoted by `R.A`), or an arithmetic expression.
– A Boolean condition `wcond` in the WHERE clause of a select statement is built up in the usual way, using also the standard comparison operators.

Below, we show a syntactic transformation $[\_]_{\mathcal{RA}}$ that maps every select statement to an equivalent *RA*-expression in the usual way[1].

– $[\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_m \text{ WHERE } \mathtt{wcond}]_{\mathcal{RA}} = \pi_{\mathtt{exp}_1,\ldots,\mathtt{exp}_k}(\sigma_{\mathtt{wcond}}(\mathtt{R}_1 \times \ldots \times \mathtt{R}_m))$
– $[\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2]_{\mathcal{RA}} = [\mathtt{sel\_stm}_1]_{\mathcal{RA}} \bigcup [\mathtt{sel\_stm}_2]_{\mathcal{RA}}$
– $[\mathtt{sel\_stm} \text{ EXCEPT } \mathtt{R}]_{\mathcal{RA}} = [\mathtt{sel\_stm}]_{\mathcal{RA}} - \mathtt{R}$

The formal meaning of every `sel_stm` w.r.t. an interpretation I, stated in Definition 5 (Section 3), evinces the idea that the expected interpretation of a select statement $[\![\mathtt{sel\_stm}]\!]^I$ should be the set of tuples associated to the corresponding equivalent *RA*-expression $[\mathtt{sel\_stm}]_{\mathcal{RA}}$.

2.2 Expressiveness of R-SQL

Next, we illustrate that R-SQL overcomes some limitations present in current RDBMS's following SQL-99. These languages use NOT EXITS and EXCEPT clauses to deal with negation, and WITH RECURSIVE to engage recursion. As it is pointed out in [5], SQL-99 does not allow an arbitrary collection of mutually recursive relations to be written in the WITH RECURSIVE clause. Although any mutual recursion can be converted to direct recursion by inlining [8], our proposal allows to explicitly define mutual recursive relations, which is an advantage in terms of program readability and maintenance. For instance, using R-SQL, it is easy to write the classical example for computing even and odd numbers up to a bound (100 in the example) as follows:

```
even(x float) := SELECT 0 UNION
  SELECT odd.x+1 FROM odd WHERE odd.x<100;

odd(x float) := SELECT even.x+1 FROM even WHERE even.x<100;
```

Further, linear recursion in standard SQL restricts the number of allowed recursive calls to be only one, i.e., Fibonacci numbers cannot be specified as follows[2]:

[1] Notice that arithmetic expressions are allowed as arguments in *projection* (π) and *select* (σ) operations.

[2] The relations `fib1` and `fib2` simply represent two aliases for `fib`, which are necessary because, for simplicity, we have not added support for renamings in R-SQL FROM clauses.

```
fib1(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib2(n float, f float) := SELECT fib.n, fib.f FROM fib;

fib(n float, f float) := SELECT 0,1 UNION SELECT 1,1 UNION
  SELECT fib1.n+1,fib1.f+fib2.f FROM fib1,fib2
  WHERE fib1.n=fib2.n+1 AND fib1.n<10;
```

This means that several graph algorithms specified using non-linear recursion cannot be directly expressed in current recursive SQL systems [17].

Non-termination is another problem that arises associated to recursion. For instance, the basic transitive closure over a graph that includes a cycle makes current SQL systems (such as PostgreSQL and MySQL) either to reject the query or to go into an infinite loop (some systems allow to impose a maximum number of iterations as a simple termination condition). Nevertheless, the fix-point computation used by R-SQL guarantees termination when dealing with finite relations. The following example written in R-SQL defines the relations `arc` (a graph with a cycle) and `path` (its transitive closure). The computation is terminating since both relations are finite.

```
arc(ori varchar(1), des varchar(1)) :=
  SELECT a,b UNION SELECT b,c UNION SELECT c,a;

path(ori varchar(1), des varchar(1)) :=
  SELECT arc.ori, arc.des FROM arc UNION
  SELECT arc.ori, path.des FROM arc,path WHERE arc.des=path.ori
```

The following running example contains a concrete relation defined using the classical transitive closure technique mentioned above.

Example 1. A database for flights. As usual, the information about direct flights can be composed of the city of origin, the city of destination, and the length of the flight. Cities (Lisboa, Madrid, Paris, London, New York) will be represented with constants (`lis`, `mad`, `par`, `lon`, `ny`, resp.)

```
flight(frm varchar(10), to varchar(10), time float) :=
  SELECT 'lis','mad',1.0 UNION SELECT 'mad','par',1.5 UNION
  SELECT 'par','lon',2.0 UNION SELECT 'lon','ny',7.0  UNION
  SELECT 'par','ny',8.0;
```

The relation `reachable` consists of all the possible trips between the cities of the database, maybe concatenating more than one flight.

```
reachable(frm varchar(10), to varchar(10)) :=
  SELECT flight.frm, flight.to FROM flight UNION
  SELECT reachable.frm, flight.to FROM reachable,flight
  WHERE reachable.to = flight.frm;
```

The relation `travel` also gives time information about alternative trips.

```
travel(frm varchar(10), to varchar(10), time float) :=
  SELECT flight.frm, flight.to, flight.time
  FROM flight UNION
  SELECT flight.frm, travel.to, flight.time+travel.time
  FROM flight, travel WHERE flight.to = travel.frm;
```

Both `reachable` and `travel` represent transitive closures of the relation `flight`. Notice that if `flight` has a cycle, then the relation `travel` that includes times for each trip is infinite, while `reachable` is not. As pointed before, `reachable` can be finitely computed in our system. But, as `travel` would produce an infinite set of different tuples, some computation limitation would have to be imposed (as the maximum time for a `travel`, for example). However, this is not a drawback of our approach, but an issue due to using infinite relations (built with arithmetic expressions).

The relation `madAirport` contains travels departing or arriving in Madrid, while `avoidMad` contains possible travels that neither begin, nor end in Madrid.

```
madAirport(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable
  WHERE (reachable.frm = 'mad' OR reachable.to = 'mad');

avoidMad(frm varchar(10), to varchar(10)) :=
  SELECT reachable.frm, reachable.to FROM reachable
  EXCEPT madAirport;
```

This definition includes negation together with recursive relations. This combination can not be expressed in SQL-99 as it is shown in [4].

3 A Stratified Fixpoint Semantics for R-SQL

It is well-known that the combination of negation and recursion in database languages is a difficult task [1]. This problem has been tackled with stratified fixpoint semantics in several works [12,11,14], and we have found that these techniques can be also applied to our proposal to obtain an operational semantics for R-SQL. In this section we present a novel formalization of recursive SQL relations by means of a stratified fixpoint interpretation that formalizes the meaning of R-SQL-databases, and we show how to compute such fixpoint.

Next, we introduce the notions of dependency graph and stratification that provide the basis for the stratified negation formalization we are looking for. Then, we define the concept of stratified interpretations, and prove the existence of the fixpoint of a continuous operator as the required interpretation of a database. The obtained semantics will be the basis of the implementation of a concrete R-SQL database system.

3.1 Dependency Graph and Stratification

Stratification is based on the definition of a *dependency graph* for a database. In the following, we consider a database `sql_db` defined as $\texttt{R}_1\texttt{sch}_1$:= $\texttt{sel\_stm}_1$; . . . ;

$\texttt{R}_n\texttt{sch}_n$:= $\texttt{sel\_stm}_n$. We denote by RN the set $\{\texttt{R}_1,\ldots,\texttt{R}_n\}$ of relation names of sql_db. We assume that relations are well defined, in the sense that the relation names used inside $\texttt{sel\_stm}_1 \ldots \texttt{sel\_stm}_n$ are in RN. The dependency graph associated to sql_db, denoted by $DG_{\texttt{sql\_db}}$, is a directed graph whose nodes are the elements of RN, and the edges, that can be *negatively labelled*, are determined by the dependencies between the database relations, that are defined as follows. A relation definition of the form `R sch := sel_stm` produces edges in the graph from every relation name inside sel_stm to R. Those edges produced by the relation name that is just to the right of an EXCEPT are negatively labelled.

Definition 1. For every two relations $\texttt{R}_1$, $\texttt{R}_2 \in \texttt{RN}$, we say:

- $\texttt{R}_2$ *depends* on $\texttt{R}_1$ if there is a path from $\texttt{R}_1$ to $\texttt{R}_2$ in $DG_{\texttt{sql\_db}}$.
- $\texttt{R}_2$ *negatively depends* on $\texttt{R}_1$ if there is a path from $\texttt{R}_1$ to $\texttt{R}_2$ in $DG_{\texttt{sql\_db}}$ with at least one negatively labelled edge.

Example 2. Consider the database of Example 1. Its corresponding set of relation names is RN $= \{\texttt{flight}, \texttt{reachable}, \texttt{travel}, \texttt{madAirport}, \texttt{avoidMad}\}$. Its dependency graph is depicted in Figure 2, where negatively labelled edges are annotated with $\neg$.

Definition 2. A *stratification* of sql_db is a mapping str : RN $\to \{1,\ldots,n\}$, such that:

- $str(\texttt{R}_i) \leq str(\texttt{R}_j)$, if $\texttt{R}_j$ depends on $\texttt{R}_i$,
- $str(\texttt{R}_i) < str(\texttt{R}_j)$ if $\texttt{R}_j$ negatively depends on $\texttt{R}_i$.

sql_db is *stratifiable* if there exists a stratification for it. In this case, for every R $\in$ RN, we say that str(R) is the *stratum* of R. We denote by *numstr* the maximum stratum of the elements of RN. And str(sel_stm) represents the maximum stratum of the relations included in sel_stm.

Intuitively, a relation name preceded by an EXCEPT plays the role of a negated predicate (relation) in the deductive database field. A stratification-based solving procedure ensures that when a relation that contains an EXCEPT in its definition is going to be calculated, the meaning of the inner negated relation has been completely evaluated, avoiding nonmonotonicity, as it is widely studied in Datalog [16]. The novelty lies on introducing these ideas into the field of the relational model.

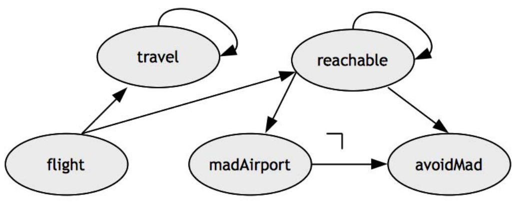


Fig. 2. $DG_{\texttt{sql\_db}}$ of Example 1

3.2 Stratified Interpretations and Fixpoint Operator

From now on, we consider a stratifiable `sql_db`, and that *str* is a stratification for it. In the previous section, we established that in a relation definition for `R sch`, the schema `sch` is a sequence of type declarations for the attributes of `R`. In order to give meaning to this relation, we assume that every type `T` included in `sch` denotes a domain D. In previous examples we have used two types: `varchar`, denoting the domain of strings, and `float`, denoting a numeric domain. We will consider a *universal domain* $\mathcal{D}$ which is the union of the family of the considered domains. Relations of arity k will denote a set of k-tuples included in $\mathcal{D}^k$. In general, every relation denotes a subset of $\mathcal{T} = \bigcup_{n\geq 1} \mathcal{D}^n$.

Interpretations are functions that associate an element of $\mathcal{P}(\mathcal{T})$ to each element of `RN`. So, considering the usual relational model terminology of schema and instance of a relation, the interpretation of a relation in our model can be seen as the relationship between the schema and the instance of the relation. Interpretations are classified by strata. An interpretation of a stratum i gives meaning to the relations of strata less or qual to i. Next, we formalize the concept of interpretation over a stratum.

Definition 3. An *interpretation* I for `sql_db`, over the stratum i, $1 \leq i \leq$ *numstr* is a function from `RN` to $\mathcal{P}(\mathcal{T})$, such that, for each $\mathtt{R} \in \mathtt{RN}$:

– If `R` has schema $(\mathtt{A}_1\mathtt{T}_1, \ldots, \mathtt{A}_r\mathtt{T}_r)$, and $D_1, \ldots, D_r$ are, respectively, the domains denoted by $\mathtt{T}_1, \ldots, \mathtt{T}_r$, then $I(\mathtt{R}) \subseteq D_1 \times \ldots \times D_r$.
– $I(\mathtt{R}) = \emptyset$, if $str(\mathtt{R}) > i$.

The set of interpretations for `sql_db` over the stratum i, $1 \leq i \leq numstr$ is denoted by $\mathcal{I}_i^{\mathtt{sql\_db}}$. The following definition provides an order on $\mathcal{I}_i^{\mathtt{sql\_db}}$.

Definition 4. Let $i \geq 1$, and $I_1, I_2 \in \mathcal{I}_i^{\mathtt{sql\_db}}$. I_1 *is less or equal than* I_2 *at stratum* i, denoted by $I_1 \sqsubseteq_i I_2$, if the following conditions are satisfied for every $\mathtt{R} \in \mathtt{RN}$:

– $I_1(\mathtt{R}) = I_2(\mathtt{R})$, if $str(\mathtt{R}) < i$.
– $I_1(\mathtt{R}) \subseteq I_2(\mathtt{R})$, if $str(\mathtt{R}) = i$.

It is straightforward to check that for any i, $1 \leq i \leq numstr$, $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$ is a poset. The main question is that when an interpretation over a stratum i increases, the set of tuples associated to the relations whose stratum is i can increase, but the sets associated to relations of smaller strata remain invariable. In addition, this poset is a cpo, as it is proved in the following lemma.

Lemma 1. For any $i \geq 1$, the pair $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$ is a complete partially ordered set. Moreover, if $\{I_n\}_{n\geq 0}$ is a chain of interpretations in $(\mathcal{I}_i^{\mathtt{sql\_db}}, \sqsubseteq_i)$, then $\hat{I}$, defined as $\hat{I}(\mathtt{R}) = \bigcup_{n\geq 0} I_n(\mathtt{R})$, is the least upper bound of $\{I_n\}_{n\geq 0}$.

Proof. It is easy to prove that $\hat{I} \in \mathcal{I}_i^{\mathtt{sql\_db}}$, and that it is an upper bound. In addition, if I is another upper bound, that implies: If $str(\mathtt{R}) < i$, $I(\mathtt{R}) = I_n(\mathtt{R})$ for every $n \geq 0$, and hence $\hat{I}(\mathtt{R}) = I(\mathtt{R})$. If $str(\mathtt{R}) = i$, $I_n(\mathtt{R}) \subseteq I(\mathtt{R})$ for every $n \geq 0$, then $\bigcup_{n\geq 0} I_n(\mathtt{R}) \subseteq I(\mathtt{R})$. Therefore $\hat{I} \sqsubseteq_i I$, by the definition of $\sqsubseteq_i$. □

The following definition formalizes the meaning of a select statement `sel_stm` in the context of a concrete interpretation I, both associated to a concrete `sql_db` database. As we pointed out before, the interpretation of a `sel_stm` will be the set of tuples associated to its corresponding RA-expression, $[\texttt{sel\_stm}]_{\mathcal{RA}}$, when the value of the involved relation names is given by I.

Definition 5. Let $i \geq 1$, and $I \in \mathcal{I}_i^{\texttt{sql\_db}}$. Let `sel_stm` be a select statement including only relation names of `RN`, such that $str(\texttt{sel\_stm}) \leq i$. We recursively define the *interpretation of* `sel_stm` *w.r.t.* I, denoted by $[\![\texttt{sel\_stm}]\!]^I$, as:

- $[\![\texttt{sel\_stm}_1 \text{ UNION } \texttt{sel\_stm}_2]\!]^I = [\![\texttt{sel\_stm}_1]\!]^I \bigcup [\![\texttt{sel\_stm}_2]\!]^I$, where $\bigcup$ stands for the set union.
- $[\![\texttt{sel\_stm} \text{ EXCEPT } \texttt{R}]\!]^I = [\![\texttt{sel\_stm}]\!]^I \setminus I(\texttt{R})$, where $\setminus$ represents set difference.
- $[\![\text{SELECT } \texttt{exp}_1, \ldots, \texttt{exp}_k]\!]^I = \{(exp_1, \ldots, exp_k)\}$, where exp_i is the mathematical evaluation of $\texttt{exp}_i$.
- $[\![\text{SELECT } \texttt{exp}_1, \ldots, \texttt{exp}_k \text{ FROM } \texttt{R}_1, \ldots, \texttt{R}_m \text{ WHERE } \texttt{wcond}]\!]^I =$ $\{(exp_1[\overline{a}/\overline{\texttt{A}}], \ldots, exp_k[\overline{a}/\overline{\texttt{A}}]) | \overline{a} \in I(\texttt{R}_1) \times \ldots \times I(\texttt{R}_m) \text{ and } wcond[\overline{a}/\overline{\texttt{A}}] \text{ is satisfied}\}$.

$\overline{\texttt{A}}$ is a sequence of attributes labelled with their corresponding relation names. More precisely, if $\texttt{A}_1^j, \ldots, \texttt{A}_{r_j}^j$ are the attributes of $\texttt{R}_j$, $1 \leq j \leq m$, then $\overline{\texttt{A}}$ represents the complete sequence $\texttt{R}_1.\texttt{A}_1^1, \ldots, \texttt{R}_1.\texttt{A}_{r_1}^1, \ldots, \texttt{R}_m.\texttt{A}_1^m \ldots \texttt{R}_m.\texttt{A}_{r_m}^m$. $exp_j[\overline{a}/\overline{\texttt{A}}]$, $1 \leq j \leq k$, is the mathematical evaluation of $\texttt{exp}_j$, after replacing the tuple $\overline{a}$ by $\overline{\texttt{A}}$. And $wcond[\overline{a}/\overline{\texttt{A}}]$ is the evaluation of the Boolean expression `wcond`, with the previous substitution.

Example 3. Consider the definitions of the relations `odd` and `even` of Section 2.2. Let us assume a concrete interpretation I such that $I(\texttt{even}) = \{(0), (2)\}$ and $I(\texttt{odd}) = \emptyset$. Hence, the interpretation of the select statement that defines the relation `odd` w.r.t. I is:
$[\![\texttt{SELECT even.x+1 FROM even WHERE even.x<100}]\!]^I = \{(\texttt{even.x+1})[a/\texttt{even.x}]$ $|(a) \in I(\texttt{even}) \text{ and } (\texttt{even.x<100})\ [a/\texttt{even.x}]\text{is satisfied}\} = \{(1), (3)\}$.

The case of the relation `even` is analogous:
$[\![\texttt{SELECT 0 UNION SELECT odd.x+1 FROM odd WHERE odd.x<100}]\!]^I =$
$[\![\texttt{SELECT 0}]\!]^I \bigcup [\![\texttt{SELECT odd.x+1 FROM odd WHERE odd.x<100 }]\!]^I = \{\texttt{(0)}\} \bigcup$
$\{(\texttt{odd.x+1})[a/\texttt{odd.x}]\ |(a) \in I(\texttt{odd}), (\texttt{odd.x<100})[a/\texttt{odd.x}] \text{ is satisfied}\} = \{\texttt{(0)}\}$.

Notice that the interpretation $\hat{I}$ defined by $\hat{I}(\texttt{even}) = \{(0), (2), \ldots, (100)\}$ and $\hat{I}(\texttt{odd}) = \{(1), (3), \ldots, (99)\}$ satisfies:
$\hat{I}(\texttt{even}) = [\![\texttt{SELECT 0 UNION SELECT odd.x+1 FROM odd WHERE odd.x<100}]\!]^{\hat{I}}$.
$\hat{I}(\texttt{odd}) = [\![\texttt{SELECT even.x+1 FROM even WHERE even.x<100}]\!]^{\hat{I}}$.

The semantics of `sql_db` will be formalized by means of an interpretation I over *numstr*, such that for every $\texttt{R} \in \texttt{RN}$, if `R sch := sel_stm` is the definition of `R` in `sql_db`, then I maps the set $[\![\texttt{sel\_stm}]\!]^I$ to `R`, as the interpretation $\hat{I}$ of Example 3 does. For every stratum i, the appropriate interpretation that gives the complete meaning to each relation of stratum i is the least fixpoint of a continuous operator over the set of interpretations of stratum i. These fixpoint interpretations are constructed sequentially from stratum 1 to *numstr*.

The fixpoint of the last stratum *numstr* provides the semantics for the whole database. Some technical lemmas are shown in order to ensure the existence of such fixpoint interpretations.

The following lemma states that the sets of tuples denoted by a select statement of a stratum i, w.r.t. two ordered interpretations, satisfy an inclusion relation that is in accordance with the order $\sqsubseteq_i$ between the two interpretations.

Lemma 2. Let $i \geq 1$, $\mathtt{R} \in \mathtt{RN}$, with $str(\mathtt{R}) \leq i$, and I_1, $I_2 \in \mathcal{I}_i^{\mathtt{sql\_db}}$, such that $I_1 \sqsubseteq_i I_2$. Then, every `sel_stm` included in the select statement that defines R holds:

– If $str(\mathtt{sel\_stm}) < i$, then $[\![\mathtt{sel\_stm}]\!]^{I_1} = [\![\mathtt{sel\_stm}]\!]^{I_2}$.
– If $str(\mathtt{sel\_stm}) = i$, then $[\![\mathtt{sel\_stm}]\!]^{I_1} \subseteq [\![\mathtt{sel\_stm}]\!]^{I_2}$.

Proof. The proof is inductive on the structure of `sel_stm`. Here, we only show the most critical case. The others are similar.

$[\![\mathtt{sel\_stm}\ \text{EXCEPT}\ \mathtt{R}']\!]^{I_1} = [\![\mathtt{sel\_stm}]\!]^{I_1} \setminus I_1(\mathtt{R}')$. According to the definition of stratification, $str(\mathtt{R}') < i$, because we are assuming that `sel_stm` EXCEPT $\mathtt{R}'$ occurs in the definition of R and $str(\mathtt{R}) \leq i$. Hence $I_1(\mathtt{R}') = I_2(\mathtt{R}')$. Now, if $str(\mathtt{sel\_stm}\ \text{EXCEPT}\ \mathtt{R}') \leq i$, then $[\![\mathtt{sel\_stm}]\!]^{I_1} \subseteq [\![\mathtt{sel\_stm}]\!]^{I_2}$, applying the induction hypothesis. Therefore $[\![\mathtt{sel\_stm}\ \text{EXCEPT}\ \mathtt{R}']\!]^{I_1} \subseteq [\![\mathtt{sel\_stm}\ \text{EXCEPT}\ \mathtt{R}']\!]^{I_2}$, with equality for the case $str(\mathtt{sel\_stm}\ \text{EXCEPT}\ \mathtt{R}') < i$. □

The following lemma underlies the proof of the continuity of the operator whose fixpoint provides the semantics of a database (it can be proved by induction on the structure of `sel_stm`).

Lemma 3. Let $i \geq 1$, $\mathtt{R} \in \mathtt{RN}$, with $str(\mathtt{R}) \leq i$, and $\{I_n\}_{n\geq 0}$ be a chain in $\mathcal{I}_i^{\mathtt{sql\_db}}$. Then, for every `sel_stm` included in the definitions of R, if $\hat{I} = \bigsqcup_{n\geq 0} I_n$, there exists $n \geq 0$, such that $[\![\mathtt{sel\_stm}]\!]^{\hat{I}} = [\![\mathtt{sel\_stm}]\!]^{I_n}$.

Next, for every i, a continuous operator T_i over the set $\mathcal{I}_i^{\mathtt{sql\_db}}$ of interpretations of stratum i is defined. Analogously to the theoretical foundations that support Datalog [16], we choose the least fixpoint of T_i, as the interpretation over i that will give meaning to the relations of stratum i. In accordance with the Knaster-Tarski theorem, this fixpoint can be obtained as the least upper bound of the chain of interpretations resulting by successively applying this operator to a minimal interpretation.

Definition 6. Let $1 \leq i \leq numstr$. The *operator* $T_i : \mathcal{I}_i^{\mathtt{sql\_db}} \longrightarrow \mathcal{I}_i^{\mathtt{sql\_db}}$ transforms interpretations over i as follows. For every $I \in \mathcal{I}_i^{\mathtt{sql\_db}}$, $\mathtt{R} \in \mathtt{RN}$:

– $T_i(I)(\mathtt{R}) = I(\mathtt{R})$, if $str(\mathtt{R}) < i$.
– $T_i(I)(\mathtt{R}) = [\![\mathtt{sel\_stm}]\!]^{I}$, if $str(\mathtt{R}) = i$ and `R sch := sel_stm` is the definition of R in `sql_db`.
– $T_i(I)(\mathtt{R}) = \emptyset$, if $str(\mathtt{R}) > i$.

This operator is proved to be monotone (it is a consequence of Lemma 2) and continuous for every i.

Lemma 4. [Monotonicity of T_i] Let $i \geq 1$ and $I_1, I_2 \in \mathcal{I}_i^{\texttt{sql\_db}}$, such that $I_1 \sqsubseteq_i I_2$. Then, $T_i(I_1) \sqsubseteq_i T_i(I_2)$.

Proposition 1. [Continuity of T_i] Let $i \geq 1$ and $\{I_n\}_{n \geq 0}$ be a chain of interpretations in $\mathcal{I}_i^{\texttt{sql\_db}}$ ($I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$). Then, $T_i(\bigsqcup_{n \geq 0} I_n) =_i \bigsqcup_{n \geq 0} T_i(I_n)$.

Proof. The proof of $\bigsqcup_{n \geq 0} T_i(I_n) \sqsubseteq_i T_i(\bigsqcup_{n \geq 0} I_n)$ is a direct consequence of the monotonicity of T_i (Lemma 4). Let us prove $T_i(\bigsqcup_{n \geq 0} I_n) \sqsubseteq_i \bigsqcup_{n \geq 0} T_i(I_n)$:

– If $str(\texttt{R}) < i$, then $T_i(\bigsqcup_{n \geq 0} I_n)(\texttt{R}) = \bigsqcup_{n \geq 0} I_n(\texttt{R})$, by the definition of T_i. Now, for every $n \geq 0$, $I_n(\texttt{R}) = T_i(I_n)(\texttt{R})$, also by definition of T_i. Therefore, $(T_i(\bigsqcup_{n \geq 0} I_n))(\texttt{R}) = (\bigsqcup_{n \geq 0} T_i(I_n))(\texttt{R})$.
– If $str(\texttt{R}) = i$, then $T_i(\bigsqcup_{n \geq 0} I_n)(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{\bigsqcup_{n \geq 0} I_n}$, by definition of T_i. And, in accordance with Lemma 3, for some $n \geq 0$: $[\![\texttt{sel\_stm}]\!]^{\bigsqcup_{n \geq 0} I_n} \subseteq [\![\texttt{sel\_stm}]\!]^{I_n}$. Now $[\![\texttt{sel\_stm}]\!]^{I_n} = T_i(I_n)(\texttt{R})$, by definition of T_i, and obviously $T_i(I_n)(\texttt{R}) \subseteq \bigcup_{n \geq 0} T_i(I_n)(\texttt{R})$, but $\bigcup_{n \geq 0} T_i(I_n)(\texttt{R}) = (\bigsqcup_{n \geq 0} T_i(I_n))(\texttt{R})$, by Lemma 1. Hence, we conclude $T_i(\bigsqcup_{n \geq 0} I_n)(\texttt{R}) \subseteq (\bigsqcup_{n \geq 0} T_i(I_n))(\texttt{R})$. □

Next, the expected result corresponding to the existence of least fixpoint stratum by stratum is shown.

Lemma 5. The operator T_1 has a least fixpoint, which is $\bigsqcup_{n \geq 0} T_1^n(\emptyset)$, where $\emptyset : \texttt{RN} \to \mathcal{P}(\mathcal{T})$ is the interpretation such that $\emptyset(\texttt{R}) = \emptyset$ for every $\texttt{R} \in \texttt{RN}$.

Proof. By the Knaster-Tarski fixpoint theorem [15], using Proposition 1. □

We will denote $\bigsqcup_{n \geq 0} T_1^n(\emptyset)$ by fix_1, i.e., fix_1 represents the least fixpoint at stratum 1. Using Example 1, Figure 3 shows the tuples corresponding to the successive applications of the operator T_1 until $fix_1(\texttt{travel})$ is obtained.

Consider now the sequence $\{T_2^n(fix_1)\}_{n \geq 0}$ of interpretations in $(\mathcal{I}_2^{\texttt{sql\_db}}, \sqsubseteq_2)$ greater than fix_1. Using the definition of T_i and the fact that $fix_1(\texttt{R}) = \emptyset$ for every R such that $str(\texttt{R}) \geq 2$, it is easy to prove, by induction on $n \geq 0$, that this sequence is a chain:

$$fix_1 \sqsubseteq_2 T_2(fix_1) \sqsubseteq_2 T_2(T_2(fix_1)) \sqsubseteq_2 \ldots, \sqsubseteq_2 T_2^n(fix_1), \ldots$$

As before, in accordance with Proposition 1, $\{T_2^n(fix_1)\}_{n \geq 0}$ has a least upper bound, $\bigsqcup_{n \geq 0} T_2^n(fix_1)$, in $(\mathcal{I}_2^{\texttt{sql\_db}}, \sqsubseteq_2)$ that is the least fixpoint of T_2 containing fix_1. We denote this interpretation by fix_2.

$T_1^n(\emptyset)(\texttt{travel})$	Set of tuples
$T_1^1(\emptyset)(\texttt{travel})$	{(lon,ny,7.0), (par,lon,2.0), (par,ny,8.0), (mad,par,1.5), (lis,mad,1.0)}
$T_1^2(\emptyset)(\texttt{travel})$	{(lis,par,2.5), (par,ny,9.0), (mad,ny,9.5), (mad,lon,3.5)}
$T_1^3(\emptyset)(\texttt{travel})$	{(lis,ny,10.5), (lis,lon,4.5), (mad,ny,10.5)}
$T_1^4(\emptyset)(\texttt{travel})$	{(lis,lon,4.5), (mad,ny,10.5), (lis,ny,11.5)}

Fig. 3. Obtaining $fix_1(\texttt{travel})$

By proceeding successively, for every i, $1 < i \leq numstr$, a chain:

$$fix_{i-1} \sqsubseteq_i T_i(fix_{i-1}) \sqsubseteq_i T_i(T_i(fix_{i-1})) \sqsubseteq_i \ldots \sqsubseteq_i T_i^n(fix_{i-1}) \ldots$$

can be defined, and a fixpoint of T_i, $fix_i = \bigsqcup_{n \geq 0} T_i^n(fix_{i-1})$, can be found.

Theorem 1. There is a fixpoint interpretation $fix : \texttt{RN} \longrightarrow \mathcal{P}(\mathcal{T})$, such that for every $\texttt{R} \in \texttt{RN}$, if `sel_stm` is the definition of R, then $fix(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix}$.

Proof. The interpretation fix we are looking for is fix_{numstr}, the least fixpoint of the operator T_{numstr}, applied to $fix_{numstr-1}$. As it has been pointed out, this fixpoint exists and verifies $fix_1 \sqsubseteq_{numstr} fix_2 \sqsubseteq_{numstr} \cdots \sqsubseteq_{numstr} fix_{numstr}$. Moreover, if $str(\texttt{R}) = i$, $1 \leq i \leq numstr$, and it is defined by the statement `sel_stm`, then $fix(\texttt{R}) = fix_i(\texttt{R}) = T_i(fix_i)(\texttt{R})$, because fix_i is the fixpoint of T_i. Now, $T_i(fix_i)(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix_i}$, by definition of T_i. We can conclude $fix(\texttt{R}) = [\![\texttt{sel\_stm}]\!]^{fix}$, trivially if $i = numstr$, or using Lemma 2, if $i < numstr$, because $fix_i \sqsubseteq_{numstr} fix$. □

Therefore, the interpretation fix defines the fixpoint semantics of `sql_db`. This semantics is the support of the database system prototype we have implemented, which is described next.

4 Implementing R-SQL

In this section we introduce a working proof-of-concept implementation for the R-SQL language that takes a set of relation definitions and outputs their meanings if a stratification can be found. More specifically, taking a stratifiable database definition in the R-SQL syntax as input, we get a SQL database (for a concrete SQL database system), that corresponds to the fixpoint semantics of the input database. If the database is not stratifiable, the system throws an error message and stops.

4.1 An Algorithm to Compute the Database Fixpoint

Let `sql_db` be the definition of a R-SQL database. In order to create the corresponding SQL database we have to generate the appropriate SQL sentences for building the expected relations, that will be eventually processed by a RDBMS. The algorithm takes `sql_db` as input, i.e., a sequence of relation definitions, $\texttt{R}_1\texttt{sch}_1$:= $\texttt{sel\_stm}_1$; ...; $\texttt{R}_n\texttt{sch}_n$:= $\texttt{sel\_stm}_n$. The computation builds the dependency graph for `sql_db`, as shown in Section 3.1, then calculates a stratification for it obtaining the sets $\mathrm{R}_1, \ldots, \mathrm{R}_{numstr}$, where R_i is the set of relations of stratum i, and finally the fixpoint is computed with the following algorithm:

```
(1)  str:=1
(2)  while str ≤ numstr do
(3)      for each R_i ∈ R_str do CREATE TABLE R_i sch_i
(4)      change := true
(5)      while change do
(6)          size := rel_size_sum(R_str)
(7)          for each R_i ∈ R_str do
(8)              INSERT INTO R_i SELECT * FROM sel_stm_i
(9)                  EXCEPT SELECT * FROM R_i;
(10)         change = (size ≠ rel_size_sum(R_str))
(11)     end while
(12)     str:=str+1
(13) end while
```

This algorithm generates for each $\mathtt{R}_i$ of `sql_db` a SQL table with the elements of $fix(\mathtt{R}_i)$. Each iteration of the external *while* at line 2 corresponds to a stratum *str*, and builds the tables of the relations of this stratum, by calculating fix_{str}. To do that, first of all an empty table with the corresponding attributes is created for every relation in the stratum *str* (line 3). Then, the iteration n of the innermost *while* at line 5 computes $T^n_{str}(fix_{str-1})$, as we will explain. For every relation $\mathtt{R}_i$ of *str*, it submits the INSERT statement at line 8. The sentence SELECT * FROM $\mathtt{sel\_stm}_i$ selects all tuples as defined by the relation $\mathtt{R}_i$ (notice that $\mathtt{sel\_stm}_i$ is a valid SQL statement). Assuming that the current database instance coincides with the value of the interpretation $T^{n-1}_{str}(fix_{str-1})$, then in accordance with Definition 5, the set of tuples that satisfy that SQL statement coincides with $[\![\mathtt{sel\_stm}_i]\!]^{T^{n-1}_{str}(fix_{str-1})}$. And this is $T^n_{str}(fix_{str-1})(\mathtt{R}_i)$, by Definition 6. The tuples already present in the table are excluded to avoid repetitions (with the EXCEPT clause at line 9). In this way, $T^n_{str}(fix_{str-1})(\mathtt{R}_i)$ is obtained for every $\mathtt{R}_i$ of stratum *str*. The expression *rel_size_sum*(R_{str}) at line 10 is equal to $\sum_{\mathtt{R}\in R_{str}} |\mathtt{R}|$, where $|\mathtt{R}|$ is the current number of tuples of the table corresponding to $\mathtt{R}$. Therefore, the variable *change* controls changes on the table sizes in order to stop the process, since *change* = *false* means that $T^n_{str}(fix_{str-1}) = T^{n-1}_{str}(fix_{str-1})$, so that fix_{str} has been reached. Then, the last iteration of the external *while* calculates fix_{numstr}, the fixpoint of `sql_db`.

4.2 A Concrete Implementation

The concrete implementation of this algorithm can be done in a number of ways. We have developed a Prolog program that processes the R-SQL input file, builds the dependency graph and the stratification (if exists), and finally produces a Python module with the code of the previous section. In fact, the external *while* at line 2 is expanded according to the number of strata, writing explicitly the corresponding code for each stratum. The *for* loop at line 7 is also expanded as we will see in Example 4. We have chosen Python as the host language mainly because is multiplatform and it provides easy connections with different database systems such as PostgreSQL, MySQL, or even via ODBC, which allows connectivity to almost any RDBMS. The additional features required for the host language are basic: Loops, assignment and basic arithmetic.

Example 4. Below, we show the result of executing our proposed algorithm for the `sql_db` of Example 1. The system assigns stratum 1 to `flight, reachable, travel, madAirport`, and stratum 2 to `avoidMad`. Next, we detail some parts of the code generated stratum by stratum. Firstly, for stratum 1, we have:

```
while change do
    size := rel_size_sum(R_str)
    INSERT INTO flight SELECT 'lis','mad',1 UNION SELECT 'mad','par',1
    UNION SELECT 'par','lon',2 UNION SELECT 'lon','ny',7
    UNION SELECT 'par','ny',8 EXCEPT SELECT * FROM flight;

    INSERT INTO reachable SELECT flight.frm, flight.to
    FROM flight UNION SELECT reachable.frm, flight.to
    WHERE reachable.to = flight.frm
    EXCEPT SELECT * FROM reachable;

    INSERT INTO travel SELECT * FROM flight UNION
    SELECT flight.frm, travel.to, flight.time+travel.time
    FROM flight, travel WHERE flight.to = travel.frm
    EXCEPT SELECT * FROM travel;

    INSERT INTO madAirport SELECT travel.frm,travel.to
    FROM travel EXCEPT SELECT * FROM madAirport;
    change = (size ≠ rel_size_sum(R_str))
end while
```

In the first iteration of this loop, we obtain all the tuples for `flight` and `madAirport` relations. But the recursive definitions for `reachable` and `travel` need more iterations. As mentioned before, those iterations correspond to the successive applications of T_1. The tuples added for `travel` at each iteration are shown in Figure 3 (Section 3.2). After five iterations, the loop stops and the first stratum is completed. In the second stratum we consider the `avoidMad` relation:

```
INSERT INTO avoidMad SELECT travel.frm,travel.to FROM travel
EXCEPT SELECT * FROM madAirport EXCEPT SELECT * FROM avoidMad;
```

This second loop ends after two iterations. This completes fix_2 for our `sql_db`, i.e., it obtains the semantics of the working example database.

4.3 Integrating R-SQL into a RDBMS

Our proposal establishes the core for introducing a novel approach for recursion in SQL. The current implementation of R-SQL has been conceived as a proof-of-concept of the theoretical foundations of the language. As we have stated, this leads to compute the semantics of the whole database from scratch. Nevertheless, the main goal of the proposal is not to introduce a new database language, but to allow less-restricted recursive relation definitions in existing SQL engines. In that sense, our proposal can be understood as the foundation of an existing SQL RDBMS that supports extended forms of recursion, allowing users to define recursive relations as regular views using the R-SQL techniques, developed in this work. Once an R-SQL database definition has been processed, the tables obtained can be stored as a database instance in a concrete RDBMS. On the one hand, the user can formulate queries that will be solved using those tables (without performing any further fixpoint computation). On the other hand, as we pointed out before, the user can define new recursive relations using views. Those views can be readily used in conjunction with other regular views, and they can be either computed on demand or can be materialized.

In order to compute the answer of new recursive relations, the current (relation) instance can be considered as a stratified R-SQL database. It is correct to assign higher strata to the new relations, because none of the existing relations depend on the new ones, and a relation definition does not introduce dependencies between the relations that appear in its select statement. Then, their tuples can be obtained by executing the algorithm in Section 4.1 to compute the fixpoint of their corresponding strata, therefore saving recalculating the previous ones. Moreover, it is straightforward to modify the algorithm to get a lazy evaluation of such relations, performing iterations only when new values are demanded. To seamlessly integrate this into a RDBMS, we can profit from the fourth-generation languages (e.g., SQL PL in IBM DB2 and PL/SQL in Oracle).

5 Conclusions

In this paper, we have introduced the R-SQL language as a new approach for incorporating recursion in SQL. This is not a trivial task, and it was not addressed in the initial proposals of SQL. It was firstly introduced in the 1999 standard, allowing only a limited form of recursion, namely linear recursion, which does not allow neither multiple recursive calls nor mutually recursive definitions. The difficulties increase when recursion is combined with negation.

We have developed a theoretical framework and a suitable implementation for R-SQL, inspired on the stratification techniques and fixpoint computations used for instance in Datalog. The stratification mechanism implies to impose some syntactic conditions on the database definitions, that guarantee that the fixpoint for such a database can be computed in a finite number of steps. This condition is less restrictive than the linearity conditions required by the standard SQL. This means that our approach is more expressive than the one adopted in SQL; in addition our language is supported by a solid computational semantics. We have presented a proof-of-concept implementation of the R-SQL database definition language based on this semantics. This implementation produces as output a set of standard SQL statements embedded in a Python program that builds the relational tables corresponding to the fixpoint of the input database definition. This implementation has been tested with PostgreSQL, but the architecture can be easily ported to any RDBMS. The system is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RSQL`.

As already suggested, our approach can be integrated into a state-of-the-art RDBMS. This can be dealt by resorting to database function definitions, which allow cursor-returning functions. In addition for this integration to be practical, performance improvements play a key role as, e.g., indexing of temporary relations during fixpoint computations and identifying tuple seeds in relation definitions that do not need to be recomputed.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Codd, E.: A Relational Model for Large Shared Databanks. Communications of the ACM 13(6), 377–390 (1970)
3. Date, C.J.: SQL and relational theory: how to write accurate SQL code. O'Reilly, Sebastopol (2009)
4. Finkelstein, S.J., Mattos, N., Mumick, I.S., Pirahesh, H.: Expressing recursive queries in SQL. Technical report, ISO (1996)
5. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems - the complete book, 2nd edn. Pearson Education (2009)
6. Houtsma, M.A.W., Apers, P.M.G.: Algebraic optimization of recursive queries. Data Knowl. Eng. 7, 299–325 (1991)
7. ISO/IEC. SQL: 2008 ISO/IEC 9075 (1-4,9-11,13,14): 2008 Standard (2008)
8. Kaser, O., Ramakrishnan, C.R., Pawagi, S.: On the conversion of indirect to direct recursion. ACM Lett. Program. Lang. Syst. 2(1-4), 151–164 (1993)
9. Kowalski, R.A.: Logic for data description. In: Logic and Data Bases, pp. 77–103 (1977)
10. Mumick, I.S., Pirahesh, H.: Implementation of magic-sets in a relational database system. SIGMOD Rec. 23, 103–114 (1994)
11. Nieva, S., Sánchez-Hernández, J., Sáenz-Pérez, F.: Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 289–304. Springer, Heidelberg (2008)
12. Ramamohanarao, K., Harland, J.: An introduction to deductive database languages and systems. The VLDB Journal 3(2), 107–122 (1994)
13. Reiter, R.: Towards a logical reconstruction of relational database theory. In: On Conceptual Modelling (Intervale), pp. 191–233 (1982)
14. Shepherdson, J.: Negation in logic programming. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 19–88. Kaufmann, Los Altos (1988)
15. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309 (1955)
16. Ullman, J.: Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies). Computer Science Press (1995)
17. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann Publishers Inc. (1997)

A Declarative-Friendly API for Web Document Manipulation

Benjamin Canou[1], Emmanuel Chailloux[1], and Vincent Balat[2]

[1] LIP6 - UMR 7606, Université Pierre et Marie Curie, Sorbonne Universités, 4 place Jussieu, 75005 Paris, France
[2] CNRS, PPS UMR 7126, Univ Paris Diderot, Sorbonne Paris Cité, F-75205 Paris, France

Abstract. The Document Object Model (DOM) is the document manipulation API provided to the JavaScript developer by the browser. It allows the programmer to update the currently displayed Web page from a client side script. For this, DOM primitives can create, remove or modify element nodes in the internal tree representation of the document. Interactive documents can be created by attaching event handlers and other auxiliary data to these nodes. The principle is interesting and powerful, and no modern Web development could be possible without it. But the implementation is not satisfactory when seeking predictability and reliability, such as expected with declarative languages or static type systems. Primitives are too generic, and when called in abnormal conditions can either throw exceptions or perform implicit imperative actions. In particular, DOM primitives can conditionally and implicitly move nodes in the document, in a way very difficult to be statically prevented or even detected. In this article, we introduce ${}^{c}DOM$, an alternative document model that performs implicit deep copies instead of moves. By not moving their children implicitly, it preserves the structure of nodes after their creation and between explicit mutations. Side data embedded in the document are also duplicated in a sensible way so that the copies are completely similar in structure to the originals. It thus provides a more usual semantics, over which existing declarative abstractions and type systems can be used in a sound way.[1]

Keywords: document manipulation, Web programming, multi-paradigm.

1 Introduction

In most widespread Web programming solutions, the programmer has to master numerous programming languages and environments. At least, she has to know HTML and CSS, the content description languages, some server language to generate pages like PHP or Perl, and the JavaScript programming language

[1] Work partially supported by the French national research agency (ANR), PWD project, grant `ANR-09-EMER-009-01`, and partly performed at the IRILL center for Free Software Research and Innovation in Paris, France.

to make Web pages interactive. Moreover, she has to handle the interactions between these languages, a task usually done with very rudimentary techniques, like directly writing client code using strings of the server language. This issue is known in the literature as an *impedance mismatch* problem. Recent Web solutions, from industry (such as Google Web Toolkit or Node.js) as well as research projects (such as Links [4], Ocsigen [1] or HOP [9]), have already tackled this problem, putting a great amount of work to give a uniform solution to program the server and the browser at language level.

However, another impedance mismatch problem remains in these solutions. The document manipulation APIs are different on the server and in the browser, making the programmer work with the same document in very different ways. In the past, this has not really been an issue. The server only produced the document, while the client updated it by inserting new document parts dynamically requested to the server. But in modern Web applications, this separation of roles is not true anymore. The server side may want to perform mutations on the document, while the client side certainly wants to create new content without asking for the server to generate it. Providing the same document manipulation API on both sides has thus become a key point for integrated client-server frameworks.

For untyped, imperative software, such as Node.js which brings JavaScript to the server, this can simply be done by using the DOM on both sides. This indeed provides a uniform API, albeit a very low level one. However, as we detail in Section 2, the unusual semantics of the DOM make it incompatible with declarative languages and advanced static type systems, neither directly as an API, nor even as a low level implementation layer. Several solutions or workarounds have already been tried to enable the use of the DOM in these contexts, but none is completely satisfactory. The solution we present in this article takes another direction by designing an alternative document model. This model, $^{C}\!DOM$, is as low level as the DOM so it can be used as a replacement. However, its semantics is much more suited to be used by high level abstractions. One of the main goals is to be able to reuse existing high levels languages and tools for XML to manipulate the document in the browser. We present the intuition behind $^{C}\!DOM$ in Section 3 and then give its formal description in Section 4, along with a few implementation details. Section 5 then presents the related works, and we finally conclude this article by presenting our future research works around the topic.

2 What Is Wrong with the DOM

This section explains why using the DOM for document manipulation is not an option both for declarative programming and static typing. Section 2.1 presents our main source of concern: implicit moves. It features an example that behaves counter-intuitively from the point-of-view of a declarative programmer. Section 2.2 presents the problems indirectly introduced by implicit moves. It explains why implicit moves make some type checking problems too difficult for existing static type systems, how they hinder even purely functional document construction, and how they impact client-server programming. Section 2.3 then makes a quick tour of existing solutions and workarounds to these problems.

2.1 Implicit Moves

In specific situations, the DOM can implicitly move nodes in the document tree. Let us examine an example which leads to such an implicit move in order to understand why and when. An initial page consists of two lists containing two items each. The elements of interest have been marked by hand with *id* attributes in order to be able to retrieve the corresponding DOM nodes from JavaScript (the standard technique). Figure 1 shows the HTML code, the rendering and a visualization of the DOM tree. Within this page, we run JavaScript code that calls DOM primitives to take the first element of the first list, and append it to the second list. The effect on the DOM is indeed the insertion of the item in the second list, but at the same time its deletion from the first list. Figure 2 shows the code and the outcome. This behaviour may appear counter-intuitive to the programmer unfamiliar with the DOM. She asked for an append operation, not a move. As a result of implicit moves, the outcome of a series of DOM operations can depend a lot on the initial state and be hard to predict.

```
... <ul id="L1">
     <li id="A">A</li>
     <li id="B">B</li>
    </ul>
```

```
... <ul id="L2">
     <li id="X">X</li>
     <li id="Y">Y</li>
    </ul> ...
```

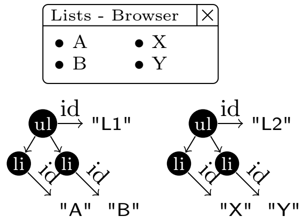


Fig. 1. A simple page

```
var l2 = getElementById ("L2")
var a = getElementById ("A")
l2.appendChild (a) ;;
```

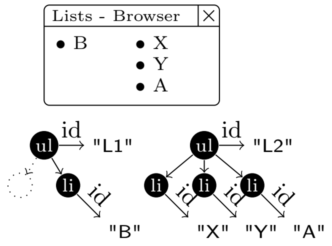


Fig. 2. An example of implicit move

Implicit moves are introduced to compensate the fact that the set of DOM primitives does not exactly fit the internal structure of the document. The memory representation of the page has to be a tree. The reason is obvious: page elements are bound to graphical objects, so cycles or sharing in the structure would not make sense. However, the set of primitives describes imperative operations on a general graph structure. A DOM primitive application that would introduce sharing or cycles in the structure is thus given an alternative behaviour that preserves the tree shape.

2.2 Side Effects of Implicit Moves

To be correctly handled by browsers, documents have to conform to precise formats. Specific, strongly typed XML processing languages such as CDuce [3] make it possible to ensure the validity of generated documents on the server. But for DOM intensive applications, verifying the validity of the initial Web page is not enough. It is often just a stub, enriched as data arrives from independent HTTP requests. It ensues that page modifications also have to be proven preserving the grammar.

Breaking validity during manipulations. Unfortunately, static checking of DOM operations is difficult, mainly because of implicit moves (which are not present in high level XML APIs). For instance, figure 3 shows a program which is well typed at first glance, since it takes references to *li* elements and adds them to an *ul* element, which is correct according to the XHTML grammar. But the outcome of executing this program with the initial (valid) Web page of figure 1 is a broken Web page, since the first list is now empty, and thus not well typed anymore. Of course this minimal example is trivial and is presented only to exhibit the problem: a DOM manipulation on one node can dynamically break the validity of another node.

```
var l2 = getElementById ("L2")
var a = getElementById ("A")
var b = getElementById ("B")
  l2.appendChild (a)
  l2.appendChild (b)
```

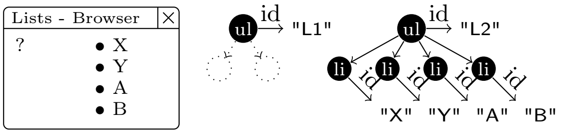


Fig. 3. A validity breaking implicit move

Breaking validity during construction. It is even possible to obtain an invalid result from a purely constructive code that would lead to a correct result using an XML language. The reason is that XML manipulation APIs allow sharing, whereas the DOM forbids sharing using implicit moves. Figure 4 shows an example HOP program, its evaluation on the server (where an XML representation that allows sharing is used) and on the client (where the back-end is the DOM). On the server, the result is the one expected by the programmer while on the client, an implicit move occurs and the first list ends up empty. Ensuring that these cases do not occur is left to the programmer. As a result, using the DOM as a back-end for high-level abstractions is not a reliable option.

HOP source

```
(let ((a (<LI> "A"))          X
      (b (<LI> "B")))         Y
  (<DIV> (<UL> (a))
         (<UL> (a b))))       Z
```

Result on the server

```
<DIV>
  <UL><LI>A</LI></UL>
  <UL><LI>A</LI><LI>B</LI></UL>
</DIV>
```

Result on the client

```
<DIV>
  <UL>          </UL>
  <UL><LI>A</LI><LI>B</LI></UL>
</DIV>
```

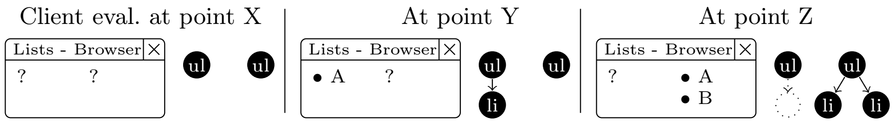


Fig. 4. Well-typed code leading to an invalid result

Influence on server document models. In integrated client-server applications, the task is not only to produce a valid document. The server also has to produce a document that will be manipulated by a client program in a valid way. In this respect, existing, server side only document models are insufficiently expressive. The problem resides in the transition between the server representation and the DOM. The technique used is always the same: nodes referenced by the server side are retrieved by the client side by searching the DOM tree for unique identifiers inserted in the XML (manually of automatically). If the server representation allows sharing, which is the case for most high level solutions, a shared node in this representation will be expanded to duplicate XML elements. Thus, any identifier it may contain will be duplicated as well, leading to undefined behaviour on the client.

2.3 Existing Solutions to Handle Implicit Moves

Of course, this problem is not new. Several solutions have been introduced by Web frameworks as well as research works. But to our knowledge, none of them offer as much versatility as the one we propose and most of them have non negligible drawbacks.

Client-server JavaScript solutions (for instance using Node.js) can use the DOM on both sides. But as we already explained, this is only an option in such untyped, imperative contexts.

High level, language based client-server frameworks (eg. HOP, Ocsigen, Links, OPA, Ur/Web) use an intermediate representation as we explained earlier. This solution has many advantages for document construction, but when it comes to document mutations, it is not better than using the DOM. The programmer has to make sure not to introduce sharing in the documents she builds, otherwise nasty side effects (including implicit moves) may occur later. Moreover, the task is actually as difficult as ensuring that no implicit move occurs with the DOM.

More mainstream frameworks hide the document structure and rely on prebuilt components instead. User code is mostly glue code, written either in server code (eg. Ruby on Rails, django) or in JavaScript (eg. ExtJS, Dojo). The main restriction is that the programmer has to compose its pages only of existing components of some framework. Frameworks are often incompatible, and writing a custom component means learning hackish internal details.

Research solutions already exist to prevent implicit moves by programs using the DOM. However, their integration to existing mainstream solutions is difficult due to the very advanced techniques used. Related research works will be detailed in Section 5.

3 An Implicitly Copying Document Model

All the existing solutions we just presented try to prevent situations which lead to implicit moves. Our solution takes a different path. ${}^{C}\!DOM$ is an alternative document model, as imperative and low level as the DOM but without implicit moves. Instead, ^{c}*DOM* performs implicit copies. It leaves the original node in place, and inserts a copy of it at destination.

3.1 Deep Copies and Auxiliary Data

To preserve document validity, deep (as opposed to shallow) copies have to be performed so the grammatical structure of original and copied nodes are the same. With existing solutions, the programmer can already detect when an implicit move can occur and replace it manually by a copy. ^{c}*DOM* simply makes this operation systematic. Recursively copying children nodes is not difficult and the DOM primitive *cloneNode* already does that very operation. But for interactive documents, such a recursive copy of nodes is not enough. To preserve the behaviour of nodes, associated auxiliary data have to be copied along with the nodes, in particular event handlers. With existing solutions, this task is manual and non trivial. It requires the programmer to make sensible decisions about what to copy, what not to and what to share with the original node. If auxiliary data are structured language values, the programmer has to decide how deep the copy has to be. Moreover, in the case of static typing, the copy operation has to preserve the types of original auxiliary data. For all these reasons, it is not possible to provide a generic deep copy algorithm for the DOM. Figure 5 shows the definition of a node and two examples of non trivial decisions to be made during a copy of this node.

```
var cpt = new Counter (0);
var node = document.createElement ("button");
node.appendChild (document.createTextNode("0"));
node.onclick = function () {
  cpt.incr (); ←——— Should a copy of node use the same counter cpt or a copy?
  var lbl = cpt.stringValue();
  node .appendChild (document.createTextNode(lbl));
}   ↑__________________ Should a copy of node modify itself or the original?
```

Fig. 5. Different options for the deep copy of a node

3.2 A Sensible Deep Copy Algorithm

In the previous example, the lack of convention made it hard to decide whether side data had to be copied or not. Providing a usable copy mechanism thus means providing such a convention, preferably an intuitive one. The major contribution of this work is thus the convention we propose, which is as follows: (1) let the programmer decide whether side data are associated to some node or not, and (2) reuse the familiar notion of lexical scope to materialize this choice. In practice, the idea is to introduce a clearly delimited syntactic construction for node definition, and use it to delimit the set of values to be copied along. Figure 6 gives an example written in such a high level (here ML based) language. In this example, if a copy of the node occurs, in both cases the callback will be copied along and act on the copied node rather than the original. However, in

With shared reference

```
let with_shared_counter =
 let r = ref 0 in (* outside *)
  let rec self =
   node <a>
    [ node <text> content = "incr" end ]
    prop on_click = fun () ->
     r := !r + 1 ;
     replace self
       [ node <text> ()
           content = string_of_int !r
         end ]
   end
  in self ;;
```

With local references

```
let with_copied_counter =
 let rec self =
  node <a>
   let r = ref 0 in (* inside *)
    [ node <text> content = "incr" end ]
    prop on_click = fun () ->
     r := !r + 1 ;
     replace self
       [ node <text> ()
           content = string_of_int !r
         end ]
  end
 in self ;;
```

Fig. 6. Self modifying graphical counter in an ML frontend to ^{c}*DOM*

one case it will use a shared counter and in the other a local copy, depending on the location of its definition.

This mechanism is more predictable, not only by the programmer but also by tools, in particular type systems. With implicit copies, the implicit mutation of nodes content is now gone. The only times when a node is modified are its creation and explicit modification. These cases can be handled by type checking the new assigned content, for instance with existing type systems for XML. The only additional restriction is that the node should not be used until it is completely built, so that a copy of an incomplete node cannot occur.

We chose not to limit our solution to a specific high level language, but to build a foundation on top of which various languages and abstractions could be built or ported. The solution we propose is to add meta-information to nodes directly in the low-level document model. This information is used as an oracle for a generic copy algorithm to decide which objects are to be copied along with the node. As we just explained, these meta-information can be used to maintain lexical scoping information at run-time. However, ^{c}*DOM*'s meta information storage is actually flexible enough for meta-information to be used in other ways, for instance to be adapted to language not equipped with a clear lexical scoping.

4 Formal Specification of ^{c}*DOM*

As the DOM, ^{c}*DOM* takes the form of a language independent API. However, ^{c}*DOM* is specified more formally by an operational semantics. This section first gives precise yet informal definitions of the concepts and then the specification.

4.1 Main Concepts

– *Document* The main concept we are formalizing is the *document* as used in the Web (eg. XML, DOM). A document assembles *nodes* that can represent textual content, graphical and semantic elements in a hierarchical structure.
– *Node* A node can have *children* nodes, and can have (at most one) *parent* node. It has a *tag* which defines its role in the document. Several nodes can have the same tag in a document. This role is not defined by the document

itself but by the program interpreting it (for instance, a Web browser will render a bullet list when it encounters a *ul* tag). The formalism presented stays at the DOM level in this sense: it is a uniform representation and does not bear any grammar notion.

- *Values* To handle textual content and programmable interactions, document nodes are enriched with side data. *Values* is the term we use to designate both nodes and side data. We distinguish *immediate* data such as integers and strings from structured data that we call *blocks* (in JavaScript, blocks designate language objects, including functions). The relation between these types is the following: $Value = Imm \cup Object$, $Object = Block \cup Node$.
- *Properties* Nodes and side data are linked using *properties*: associations between *objects* and *values* labeled with *keys*. Unlike nesting, properties can lead to sharing and cycles in the document.
- *Imperative Document* The *document* notion we just defined is inherently static, and thus not appropriate to formalize the DOM. We define the notion of *imperative document* combining a *document* as previously defined that we call the *state* with a set of *primitives* to manipulate it.
- *Primitives* To implement this separation, ${}^{C}DOM$ is specified as a set of *primitives*, an API, much as the DOM. They take parameters and return results, which are *values* as specified earlier.

4.2 Parameters

In the previous section, we described the main concepts and associated types provided by our formal model. These definitions are made more flexible by defining some of the notions as parameters, so they are not fixed by the model but are to be instantiated specifically for each implementation.

- *Tag* The set of possible node tags. There are no constraints on this parameter for the semantics to be sound but a specific implementation may add some.
- *Imm* The (unrestricted) domain of immediate values.
- *Key* The domain of object property names. It has to be enumerable and provided with a total order. In practice, keys have to be immutable.
- *Nil* The type of unimportant values. In this formalism, Nil is not an implicit subtype of everything. Types that contain Nil will be written as such.
- *Int* The representation of integers. The formalism relies on the mathematical definition, but in practice, there is no chance for a document to contain a node with a number of children that would trigger computer arithmetic overflows, so the approximation is reasonable.
- $Enum(S)$ Some primitives return not only one but a collection of results. $Enum(S)$ is the representation of collections of elements of a type S. In the semantics, the transition between mathematical sets and concrete collections is exhibited by the use of the function $enum : \mathcal{P}(S) \to Enum(S)$.

4.3 Document State

The document state is specified in ${}^{C}DOM$ by a tuple (H, L, T, P, S, s). Letters are mnemonics for Heap, Labels, Tree, Properties, Scopes and Stack. The first four components (H, L, T, P) describe the document structure while the last two (S, s) describe the dynamic scoping information.

- $H \subseteq \mathit{Node} \cup \mathit{Block}$ is the domain of existing objects.
- $L \subseteq \mathit{Node} \times \mathit{Tag}$ gives a tag to each node of the document.
- $T \subseteq \mathit{Node} \times \mathit{List}(\mathit{Node})$ associates to each node the list of its children.
- $P \subseteq \mathit{Object} \times \mathit{Key} \times \mathit{Value}$ associates objects to values through labels.
- $S \subseteq \mathit{Node} \times \mathit{Object}$ records for each nodes the objects under its scope.
- $s \in \mathit{List}(\mathit{Node})$ represents the stack of currently opened scopes.

We intentionally chose a simple mathematical structure to ease implementation, and be close to data structures. But this structure is not precise enough to express the document structure. We thus restrict it using the following *well-formed* predicate. A notable point is that this predicate is only useful at specification level and is transparent to the implementer: well-formedness is preserved by definition of the primitives. The implementer only has to correctly map the specification to her data structure and the body of her primitives.

Definition 1. *A tuple (H, L, T, P, S, s) is a well-formed ^{C}DOM state if and only if (1) L maps each node in H to a unique tag (2) T is a forest (no sharing, no cycles) over $H \cap Node$ (3) T and P only reference nodes present in H (4) P only references blocks present in H (5) An object can be in the scope of only one node in S (6) No cyclic scope chain exist in S.*

Notations. To increase readability, in the following formulas and figures, $\bullet$ means a node, $\circ$ a block and $◐$ an object. Labeled versions are used when disambiguation is required (eg. $\bullet_x$, $\circ_y$). We also define operators to compute the descendants and ancestors of a node.

$$\mathit{Desc}(\bullet) = \bigcup_{\bullet' \in T(\bullet)} (\{\bullet'\} \cup \mathit{Desc}(\bullet'))$$
$$\mathit{Anc}(\bullet) = \{\bullet'\} \cup \mathit{Anc}(\bullet') \text{ if } \exists \bullet', \bullet \in T(\bullet'),\ \emptyset \text{ otherwise}$$

4.4 API

The following list gives the complete ^{C}DOM API, the parameters and result types in the form *return type* `primitive` *(types of parameters)*. ^{C}DOM primitives are divided into two main subsets: accessing (reading) primitives and modifying (writing) primitives.

- *Int* `children` (*Node*)
- *Enum*(*Node*) `roots` (*Nil*)
- *Value* + *Nil* `get` (*Object*, *Key*)
- *Node* `create_node` (*Tag*)
- *Nil* `close` (*Node*)
- *Nil* `detach` (*Node*)
- *Nil* `bind` (*Node*, *Node*)
- *Nil* `unset` (*Object*, *Key*)
- *Node* + *Nil* `child` (*Node*, *Int*)
- *Enum*(*Key*) `properties` (*Object*)
- *Tag* `tag` (*Node*)
- *Object* `create_block` (*Nil*)
- *Nil* `reopen` (*Node*)
- *Node* `copy` (*Node*)
- *Nil* `set` (*Object*, *Key*, *Value*)

Semantic rules. The behaviour of each primitive is described by a set of (for some only one) semantic rule(s). Each rule is of the form

$$\frac{\textsc{(rule)}\quad \textit{conditions}}{S \vdash \texttt{prim}(a_0, \cdots, a_n) = r, S'}$$

reading: given arguments $(a_0, \cdots, a_n)$ and an initial state S, if the *conditions* are verified, the primitive `prim` can be applied, and the rule (Rule) can be elected to describe the behaviour of this application. If so, its return value is r, and the original state is transformed into the new state S'.

Accessing primitives. Figure 7 gives the semantics of primitives which are only meant to read the document state from the host language, without modifying it. To browse the document tree, `roots` gives the root nodes (more than one document root can be present, for instance any newly created node is considered a root), `children` and `child` allow to browse the structure by respectively giving the number of children and the n^{th} child of a node. The set of properties defined by a given node is obtained with `properties`, and `get` gives the value of a given property. The tag of a node is given by `tag`.

$$\frac{\textsc{(child)}\quad \bullet \in H \cap \mathit{Node} \qquad 0 \leqslant i < \mathit{length}(T(\bullet))}{S \vdash \texttt{child}(\bullet, i) = \mathit{nth}(T(\bullet), i), S} \qquad \frac{\textsc{(children)}\, \bullet \in H \cap \mathit{Node}}{S \vdash \texttt{children}(\bullet) = \mathit{length}(T(\bullet)), S}$$

$$\frac{\textsc{(child-unbound)} \qquad \bullet \in H \cap \mathit{Node} \qquad \neg(0 \leqslant i < \mathit{length}(T(\bullet)))}{S \vdash \texttt{child}(\bullet, i) = \mathit{nil}, S}$$

$$\frac{\textsc{(roots)}}{S \vdash \texttt{roots}(\mathit{nil}) = \mathit{enum}(\{\bullet | \mathit{Anc}(\bullet) = \emptyset\}), S} \qquad \frac{\textsc{(tag)} \quad (\bullet, t) \in L}{S \vdash \texttt{tag}(\bullet) = t, S}$$

$$\frac{\textsc{(properties)} \qquad \oslash \in H}{S \vdash \texttt{properties}(\oslash) = \mathit{enum}(\{k | (\oslash, k, v) \in P\}), S}$$

$$\frac{\textsc{(get)} \quad \exists(\oslash, k, v) \in P}{S \vdash \texttt{get}(\oslash, k) = v, S} \qquad \frac{\textsc{(get-unbound)} \qquad \nexists(\oslash, k, v) \in P}{S \vdash \texttt{get}(\oslash, k) = \mathit{nil}, S}$$

Fig. 7. Semantics of accessing primitives

Modifying primitives. Figure 8 gives the semantics of primitives that modify the document state. For block related primitives, `create_block` allocates a new, empty one, `set` either creates or assigns a property depending on its preexistence and `unset` removes a property. For node related primitives, `create_node` allocates a fresh one, `detach` removes the link between a node and its parent, and `bind` links a node to a parent. Two rules describe the evaluation of the later: either (1) the node is simply attached to its new parent if it is a root and if the new link does not create a cyclic chain in T, or (2) a deep copy of the node is performed by delegation to the explicit `copy` primitive and the result is attached to the parent.

Scope information. When a new node is allocated, its scope is automatically opened on the scope stack. Scopes are explicitly closed, using the `close` primitive. We also added a `reopen` primitive to push again on the scope stack a node whose scope has already been closed. This primitive may or may not be necessary, depending on the high level primitives given to the programmer. For instance, in an object oriented language, it may be sensible to consider a method call on

$$\textbf{(SET)}\ \dfrac{v \in H \cup Imm \qquad k \in Key \qquad \LEFTcircle \in H \qquad \nexists v', (\LEFTcircle, k, v') \in P}{(H, L, T, P, S, s) \ \vdash \ \texttt{set}(\LEFTcircle, k, v) = nil, (H, L, T, P \cup (\LEFTcircle, k, v), S, s)}$$

$$\textbf{(MODIFY)}\ \dfrac{v \in H \cup Imm \qquad \exists v' \left(\LEFTcircle, k, v'\right) \in P}{(H, L, T, P, S, s) \ \vdash \ \texttt{set}(\LEFTcircle, k, v) = nil, \left(H, L, T, P \backslash \{(\LEFTcircle, k, v')\} \cup \{(\LEFTcircle, k, v)\}, S, s\right)}$$

$$\textbf{(UNSET-1)}\ \dfrac{\exists \left(\LEFTcircle, k, v\right) \in P}{(H, L, T, P, S, s) \ \vdash \ \texttt{unset}(\LEFTcircle, k, v) = nil, (H, L, T, P \backslash \left\{(\LEFTcircle, k, v)\right\}, S, s)}$$

$$\textbf{(UNSET-2)}\ \dfrac{\nexists \left(\LEFTcircle, k, v\right) \in P}{(H, L, T, P, S, s) \ \vdash \ \texttt{unset}(\LEFTcircle, k, v) = nil, (H, L, T, P, S, s)}$$

$$\textbf{(CREATE}^{\bullet}\textbf{)}\ \dfrac{\bullet_n \notin H}{(H, L, T, P, S, \bullet_p :: s) \ \vdash \ \texttt{create\_node}(nil) = \bullet_n, (H \cup \left\{\bullet_n\right\}, L, T, P, S \cup \left\{(\bullet_p, \bullet_n)\right\}, \bullet_n :: \bullet_p :: s)}$$

$$\textbf{(CREATE}^{\circ}\textbf{)}\ \dfrac{\circ_n \notin H}{(H, L, T, P, S, \bullet_p :: s) \ \vdash \ \texttt{create\_block}(nil) = \circ_n, (H \cup \left\{\circ_n\right\}, L, T, P, S \cup \left\{(\bullet_p, \circ_n)\right\}, \bullet_p :: s)}$$

$$\textbf{(CREATE-ROOT}^{\bullet}\textbf{)}\ \dfrac{\bullet_n \notin H}{(H, L, T, P, S, [\,]) \ \vdash \ \texttt{create\_node}(nil) = \bullet_n, (H \cup \left\{\bullet_n\right\}, L, T, P, S, \bullet_n :: [\,])}$$

$$\textbf{(CLOSE-SCOPE)}\ \dfrac{}{(H, L, T, P, S, \bullet_p :: s) \ \vdash \ \texttt{close}(nil) = nil, (H, L, T, P, S, s)}$$

$$\textbf{(CREATE-ROOT}^{\circ}\textbf{)}\ \dfrac{\circ_n \notin H}{(H, L, T, P, S, [\,]) \ \vdash \ \texttt{create\_block}(nil) = \circ_n, (H \cup \left\{\circ_n\right\}, L, T, P, S, [\,])}$$

$$\textbf{(REOPEN-SCOPE)}\ \dfrac{\bullet_p \in H \cap Node}{(H, L, T, P, S, s) \ \vdash \ \texttt{reopen}(\bullet_p) = nil, (H, L, T, P, S, \bullet_p :: s)}$$

$$\textbf{(DETACH-1)}\ \dfrac{\exists \bullet_p \in H \cap Node, \bullet_n \in T(\bullet_p)}{(H, L, T, P, S, s) \ \vdash \ \texttt{detach}(\bullet_n) = nil, (H, L, T \backslash \{(\bullet_p, l)\} \cup \{(\bullet_p, l - \bullet_n)\}, P), S, s}$$

$$\textbf{(DETACH-2)}\ \dfrac{\bullet_n \in H \cap Node \qquad Anc(\bullet_n) = \emptyset}{(H, L, T, P, S, s) \ \vdash \ \texttt{detach}(\bullet_n) = nil, (H, L, T, P, S, s)}$$

$$\textbf{(ATTACH)}\ \dfrac{\bullet_p \in H \cap Node \qquad \bullet_n \in H \cap Node \qquad Anc(\bullet_n) = \emptyset \qquad \bullet_n \notin Anc(\bullet_p)}{(H, L, T, P, S, s) \ \vdash \ \texttt{bind}(\bullet_p, \bullet_n) = nil, (H, L, T\left[\bullet_p \to \bullet_n :: T(\bullet_p)\right], P, S, s)}$$

$$\textbf{(ATTACH-COPY)}\ \dfrac{\begin{array}{c}\bullet_p \in H \cap Node \qquad \bullet_n \in H \cap Node \qquad Anc(\bullet_n) \neq \emptyset \vee \bullet_n \in Anc(\bullet_p) \\ (H, L, T, P, S, s) \ \vdash \ \texttt{copy}(\bullet_n) = \bullet_{n'}, \left(H', T', P', S', s\right)\end{array}}{(H, L, T, P, S, s) \ \vdash \ \texttt{bind}(\bullet_p, \bullet_n) = nil, \left(H', L', T'\left[\bullet_p \to \bullet_{n'} :: T'(\bullet_p)\right], P', S', s\right)}$$

Fig. 8. Semantics of modifying primitives

a node as within its scope. Every new object alocated by one of the `create` primitives is associated in S to the top node present in the scope stack s. If the scope stack is empty, the new object is considered not associated to any node.

Copy. ^{c}DOM provides an explicit `copy` primitive which takes a node $\bullet$ and returns its deep copy $\bullet'$. Descendent nodes are duplicated unconditionally so that no sharing or cycle can occur, and blocks are copied or not depending on scope information. The links between nodes are copied as well, so that the duplicated value has the same memory shape than the original. We will not elaborate on that matter which is a bit out of scope, but as explained earlier, the idea is of course to ensure that both can be considered of the same type, so that the model is usable with strongly typed languages. Let us start with an intuitive description, using the graphical example of figure 10. In (1) and (2) The programmer calls `copy` on a node $\bullet$. The set of objects to copy is composed of all the objects which are reachable from $\bullet$ using both the tree structure or

$$\dfrac{\begin{array}{rl} & H' = H \cup \{\oslash | (\cdot, \oslash) \in C\} \\ & L' = L \cup \big\{(\bullet', l) \big| (\bullet, \bullet') \in C, (\bullet, l) \in L\big\} \\ & T' = T \cup \big\{(\bullet', l') \big| (\bullet, \bullet') \in C, l = T(\bullet), l' = map(rebind, l)\big\} \\ & \quad \text{where } rebind(\oslash \in H) = \oslash' \text{ if } (\oslash, \oslash') \in C, \oslash \text{ otherwise} \\ & P' = P \cup \big\{(\oslash', k, v') \big| (\oslash, \oslash') \in C, v' = rebind(P(\oslash, k))\big\} \\ \textbf{(copy)} & S' = S \cup \big\{(\bullet', \oslash') \big| (\bullet, \bullet') \in C, (\oslash, \oslash') \in C \cup C\big\} \end{array}}{(H, L, T, P, S, s) \ \vdash \ \texttt{copy}(\bullet_n) = A^{\bullet}(\bullet_n), \big(H', L', T', P', S', s\big)}$$

$$\begin{array}{rl} \text{with } C = & \{(\oslash, \oslash') | \oslash \in R, \oslash' \notin H \text{ (fresh node/block)}\} \\ \text{where } R = & fix(Restrict, \{\bullet_n\}) \ / \ Restrict(E) = \bigcup_{\bullet \in E} Desc(\bullet) \cup \bigcup_{\oslash \in E} \{\oslash' | (\oslash, \cdot, \oslash') \in P \wedge \oslash' \in I\} \\ \text{and } I = & fix(Collect, \{\bullet_n\}) \ / \ Collect(E) = \bigcup_{\bullet \in E} Desc(\bullet) \cup \bigcup_{\oslash \in E} \{\oslash' | (\oslash, \oslash') \in S\} \end{array}$$

Fig. 9. Semantics of ^{c}DOM copy operation

properties, and scope information. In (3) Selected nodes are copied by creating fresh nodes in H. In (4) and (5) The parenting links between original nodes are replicated between the duplicates, so that the forest structures of the two groups are the same. All the properties of original blocks are replicated, and the associated values are as follows. If the value is a duplicated node or block its copy is used. Otherwise the value is used as is. In the end (6), all the objects reachable from and in the scope of $\bullet$ are duplicated, the internal links between duplicated objects reflect the structure of the originals, and external links are duplicated as is. Figure 9 gives the formal semantics of the `copy` primitive. The resulting state is the original state augmented with the objects and links resulting from the copy. For this, the rule premises involve a set C of associations between copied objects and their copies. The specification of C is decomposed into the following three steps. First, we collect the set I of all the objects which are descendants or under the scope of $\bullet$, taking care of potential nested scopes. The *Collect* function describes one step of the traversal and its iteration to a fix-point gives the complete collection. We then extract from I the subset R of objects which are reachable from $\bullet$ through the document tree or properties. Finally, C associates original objects to fresh copies.

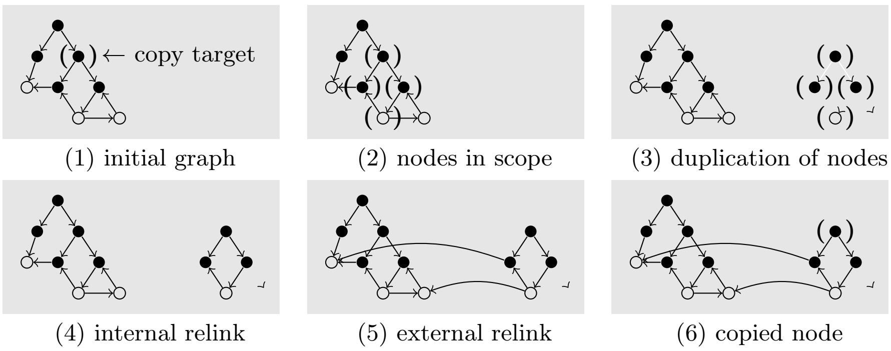


Fig. 10. Illustrated example of a copy operation

4.5 Consistency

To ensure that the API specification is sound and actually corresponds to our needs of modeling the behaviour of the DOM, we have to state that it is deterministic (in other words that it does not bypasses DOM behaviour traits by introducting indeterminism) and that it preserves the well formedness throughout primitive applications, in particular that the copy operation preserves a DOM-like structure.

Definition 2. *A primitive application is deterministic if at most one rule can be selected to describe it.*

Theorem 41 (Determinism). *For one initial state and choice of arguments, at most one rule can be elected to describe the behaviour of a given primitive application.*

Proof. *For each primitive, by showing that any pair of associated rules have mutually exclusive conditions.* □

Theorem 42 (Well-formedness preservation). *A well defined application in a well formed initial state results in a well-formed final state.*

Proof. *For most rules, simply by observing that the required conditions are a sufficient subset of the well-formed predicate clauses. The difficulty resides in the copy operation. We have to prove that all the clauses of the validity predicate are verified over the final state of the copy operation. For this, we observe that every component X' of the resulting state is the union of the original component X and a new set X^+, and that X and X^+ are always disjoint. The union of two forests over disjoint sets of nodes being also a forest, H and H^+ being disjoint, and T^+ being a forest over H^+ (because it is a copy of a subforest of T over H), we have that T' is also a forest. The same reasoning can be used to show that the structural restrictions over S' (no sharing and no cycles) are respected. Finally, a proof that the added properties only reference existing objects is obtained by definition of the rebind function, used to give values to properties in P' using only values of H and H^+.* □

We have proven that the copy operation does not break the model, now we have to prove that it is indeed useful. For this, we define a notion of similarity of structure between two nodes, and prove that the copy operation preserves the structure.

Definition 3. *Two nodes are structurally similar iff (1) they have the same tag, (2) they have the same number of children, and their children are structurally similar pairwise, and (3) They have the same set of properties, and the associated values are structurally similar pairwise. Two objects are the same if they have the same set of properties, and the associated values are structurally similar pairwise. Two immediate values are similar if they are equal.*

Theorem 43 (Structure preservation). *The node resulting of a copy operation is structurally similar to the the original.*

Proof. *By definition of the copy operation, we have directly the respect of tags, number of children and set of properties for all objects duplicated in the copy. It remains to prove that children and properties' values are similar pairwise. For each of these pairs* (v, v') *where* v *is the original and* v' *the duplicate, by definition of rebind, either* v' *is* v *or* v' *is a copy of* v *using the same copy definition. For the first case, we use the fact that* v *is structurally similar to itself. The second case is then proven by coinduction.* □

4.6 Implementing ^{c}DOM

The difficulty of implementing ^{c}DOM comes from scope information, which cannot remain static. It has to be managed at run time. We wrote two (non distributed) research prototypes using two techniques to store the scope information. This section discusses these possibilities.

The first possibility is to store a list in each allocated node, initially empty and dynamically filled with pointers to all the objects allocated within the node's scope. The copy algorithm can remain close to the specification: (1) build transitively the set of objects in the scope starting from the root to copy, (2) traverse the graph, duplicating encountered objects and links, memoizing already copied objects to respect sharing and cycles, and (3) stop following links when they point out of the set built in the first step. The memory overhead is very localized, implying no memory overhead on programs which do not perform document manipulations. By dynamically switching the allocator when not in the scope of any node, there can also be no performance overcost for such programs. Implementing this methods requires an advanced memory mechanism such as weak references or a node specific garbage collection in order not to consider alive forever any value allocated within the scope.

The second possibility is to store in each allocated object a backpointer to the node whose scope it belongs to. The algorithm is a little more complex here, because one cannot easily compute the objects in the scope of a node, apart from traversing the whole memory graph. The method is to build the deep copy by steps, maintaining a set of already copied nodes. At each step, (1) traverse the leaves of the already copied subtree and duplicate nodes and blocks backpointing to an already copied subtree, (2) traverse again the subtree, update pointers considered external at the previous step but now pointing to copied objects, redirecting them to the copies, and (3) iterate until a fix point is reached. For this method too, obtaining exact memory collection is doable only with weak pointers, but the memory leak is much more reasonable. It only arises when a node local value is put in a global reference, and not for any local value. It is thus the technique to choose to implement ^{c}DOM over current JavaScript implementations. With both methods, having a memory exact JavaScript implementation of ^{c}DOM over the DOM implies writing a garbage collection helper function, which browses the document regularly and unlinks unused objects using scope information, enabling the next JavaScript collection to actually delete them. Anyway, this will not remain a major concern for long. Weak references are already present in some browsers and are planned for a forthcoming ECMAScript specification.

5 Related Works

Works on DOM calls verification. The most advanced theoretical work has been led by Peter Thiemann [10], who proposed to integrate an ad-hoc type system into a general purpose language to check DOM calls in order to refuse statically programs which could result in implicit moves. In the same vein, there have been works on automatic tests generation to reject erroneous DOM transactions [7]. This approach is indeed a possible way to solve the problem of implicit moves, and gives direct solutions to the theoretical problems explained in section 2. However, we have two main concerns, which led us to propose our alternative approach.(1) This kind of checks cannot be directly encoded in type systems or tools available in general purpose languages. Moreover, the types or test cases to produce are complex, so these works rely on automatic solvers. Both these aspects imply practical difficulties to obtain good integration to languages and environments, in particular the difficulty to produce useful error messages and debugging possibilities. (2) This solution rejects programs that appear intuitively correct to the declarative programmer and are accepted by advanced XML functional languages such as CDuce. We thus chose to orient our solution on an alternative document model which accepts such programs.

Works on DOM specification. There have been several efforts to formalize the different components of the Web browser. We can cite the formal specification of the now defunct JavaScript 2.0 [8], a minimal formal model of JavaScript [6] or closer to our work a semantics of DOM primitives [5]. We shall not elaborate on these works, because we take an alternative approach: the formal model we develop in this paper is a simplification of the DOM.

Deep copy of DOM nodes. Libraries such as jQuery define smart copy operations able to duplicate auxiliary data but only to a limited extent. In particular, event handlers are cloned but their environment is copied only in a shallow way. Hence, the copied node will react to events, but the associated action has a good chance to be performed on the original node instead of the duplicate. It is possible to work around this behaviour by being very careful about what ends up in the environment of the event handler closure. This means having a great knowledge of the language and writing trickier code, such as flattening all the environment by hand in the DOM node, so a shallow copy will suffice, assuming that the event code only accesses its environment in an indirect way through the node.

6 Conclusion, Ongoing and Future Works

This article has presented ^{c}DOM, an alternative to the DOM. The implicit moves of the DOM are replaced by implicit deep copies that take into account auxiliary data, including event handlers. As a result, ${}^{c}DOM$ is more suited than the DOM for contexts in which predictability is important such as declarative languages or static type systems. In particular, existing functional XML languages and type systems can be ported without major modifications to the browser.

Of course, we want to prove our approach correct and usable by experimentation. For this, we have specified [2] and are currently implementing an ML-based language on top of ${}^{c}DOM$ (the one shown in one of the examples), which can run over JavaScript and the DOM, and brings static typing of document manipulations.

References

1. Balat, V., Chambart, P., Henry, G.: Client-server Web applications with Ocsigen. In: WWW 2012 Dev Track Proceedings, Lyon, France, pp. 1–4 (April 2012)
2. Benjamin, C.: Programmation Web Typée (Typed Web Programming). PhD thesis, Université Pierre et Marie Curie (2011), `http://www.pps.jussieu.fr/~canou/these.pdf` (in French)
3. Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-Centric General-Purpose Language. In: 8th International Conference on Functional Programming (ICFP 2003), pp. 51–63. ACM, New York (2003)
4. Cooper, E., Lindley, S., Yallop, J.: Links: Web Programming Without Tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)
5. Gardner, P.A., Smith, G.D., Wheelhouse, M.J., Zarfaty, U.D.: Local Hoare reasoning about DOM. In: 27th Symposium on Principles of Database Systems (PODS 2008), pp. 261–270. ACM, New York (2008)
6. Guha, A., Saftoiu, C., Krishnamurthi, S.: The Essence of JavaScript. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 126–150. Springer, Heidelberg (2010)
7. Heidegger, P., Bieniusa, A., Thiemann, P.: DOM Transactions for Testing JavaScript. In: Bottaci, L., Fraser, G. (eds.) TAIC PART 2010. LNCS, vol. 6303, pp. 211–214. Springer, Heidelberg (2010)
8. Herman, D., Flanagan, C.: Status report: specifying javascript with ML. In: Workshop on ML (ML 2007), pp. 47–52. ACM, New York (2007)
9. Serrano, M., Gallesio, E., Loitsch, F.: Hop, a language for programming the web 2.0. In: 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006), pp. 975–985. ACM, New York (2006)
10. Thiemann, P.: A Type Safe DOM API. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 169–183. Springer, Heidelberg (2005)

Implementing Equational Constraints in a Functional Language

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. KiCS2 is a new system to compile functional logic programs of the source language Curry into purely functional Haskell programs. The implementation is based on the idea to represent the search space as a data structure and logic variables as operations that generate their values. This has the advantage that one can apply various, in particular, complete search strategies or even user-defined strategies to compute solutions. However, the generation of all values for logic variables might be inefficient for applications that exploit constraints on partially known values. To overcome this drawback, we propose new techniques to implement equational constraints in this framework. In particular, we show how unification modulo function evaluation and functional patterns can be added without sacrificing the efficiency of the kernel implementation.

1 Introduction

Functional logic languages combine the most important features of functional and logic programming in a single language (see [6,16] for recent surveys). In particular, they provide higher-order functions and demand-driven evaluation from functional programming together with logic programming features like non-deterministic search and computing with partial information (logic variables). This combination has led to new design patterns [3,7] and better abstractions for application programming, but it also gave rise to new implementation challenges.

In order to implement a functional logic language, one can develop a suitable abstract machine and implement it in some (typically, imperative) language, like C [24] or Java [8,20]. One could also compile into logic languages like Prolog and reuse existing backtracking implementations for non-deterministic search as well as logic variables and unification for computing with partial information [2,23]. More recent approaches [10,12,13] compile functional logic programs into non-strict functional programs to reuse the implementation of lazy evaluation and higher-order functions. Although this requires the implementation of non-deterministic computations in a deterministic language, it has the advantage that the explicit handling of non-determinism allows for various search strategies, like depth-first, breadth-first, parallel, or iterative deepening, instead of committing to a fixed (incomplete) strategy like backtracking [12].

This paper is related to the latter implementation approach. In particular, we consider KiCS2 [11], a new system that compiles functional logic programs of the source language Curry [21] into purely functional Haskell programs. KiCS2 is based on the idea to represent the search space, i.e., all non-deterministic results of a computation, as a data structure that can be traversed by operations implementing various strategies. Logic variables are replaced by generators, i.e., operations that non-deterministically evaluate to all possible ground values of the type of the logic variable. This is justified by the fact that computing with logic variables by narrowing [28,31] and computing with generators by rewriting are equivalent, i.e., yield the same values [5]. Although this implementation technique outperforms other implementations of Curry on deterministic programs and can compete with them on non-deterministic programs (see [11] for benchmarks), the generation of all values for logic variables might be inefficient for applications that exploit constraints on partially known values. For instance, the equality constraint "`X=c(a)`" is solved in Prolog by instantiating the variable `X` to `c(a)`, but the equality constraint "`X=Y`" is solved by binding `X` to `Y` without enumerating any values for `X` or `Y`. In order to obtain a similar behavior in KiCS2, we propose in this paper new techniques to implement equational constraints (in contrast to Prolog, Curry performs unification modulo function evaluation) in a purely functional target language. A purely functional, i.e., side-effect free, implementation is reasonable in order to support different, in particular, parallel or user-defined search strategies. Beyond equational constraints, we also show how functional patterns [4], i.e., patterns containing evaluable operations for more powerful pattern matching than in logic or functional languages, can be implemented in this framework. We show that both extensions lead to efficiency improvements without sacrificing the efficiency of the kernel implementation.

In the next section, we review the source language Curry and the features considered in this paper. Section 3 recapitulates the implementation scheme of KiCS2 originally presented in [11]. Sections 4 and 5 discuss our new extensions to implement unification modulo function evaluation and functional patterns, respectively. Benchmarks demonstrating the usefulness of these extensions are presented in Sect. 6 before we conclude in Sect. 7.

2 Curry Programs

The syntax of the functional logic language Curry [21] is close to Haskell [27], i.e., type variables and names of defined operations 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 allows free (logic) variables in conditions and right-hand sides of defining rules. Hence, an operation is defined by conditional rewrite rules of the form:

$$f\, t_1 \ldots t_n \;\mid\; c \;=\; e \quad \texttt{where}\ vs\ \texttt{free} \tag{1}$$

where the *condition* c is optional and vs is the list of variables occurring in c or e but not in the *left-hand side* $f\ t_1 \ldots t_n$.

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
```

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 semantical ambiguity might occur. Consider the operations

```
not True  = False            xor True  x = not x
not False = True             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   →  not 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 prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, the rewriting logic CRWL is proposed in [15] 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 [22], where values of the arguments of an operation are determined before the operation is evaluated. In a lazy strategy, this can be enforced by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of `aBool` are shared so that all of them reduce either to `True` or to `False` consistently.

The condition c in rule (1) typically is a conjunction of *equational constraints* of the form $e_1 =:= e_2$. Such a constraint is satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms. For instance, if the symbol "`++`" denotes the usual list concatenation operation, we can define an operation `last` that computes the last element `e` of a non-empty list `xs` as follows:

```
last xs | ys++[e] =:= xs  = e   where ys, e free
```

Like in Haskell, most rules defining functions are *constructor-based* [26], i.e., in (1) $t_1, \ldots, t_n$ consist of variables and/or data constructor symbols only. However, Curry also allows *functional patterns* [4], i.e., t_i might additionally contain calls to defined operations. For instance, we can also define the last element of a list by the more concise definition

```
last' (xs++[e]) = e
```

Here, the functional pattern `(xs++[e])` states that `(last' t)` is reducible to `e` provided that the argument `t` can be matched against some value of `(xs++[e])` where `xs` and `e` are free variables. By instantiating `xs` to arbitrary lists, the value of `(xs++[e])` is any list having `e` as its last element. Functional patterns are a powerful feature to express arbitrary selections in term structures. For instance, they support a straightforward processing of XML data with incompletely specified or evolving formats [17].

3 The Compilation Scheme of KiCS2

To understand the extensions described in the subsequent sections, we review the translation of Curry programs into Haskell programs as performed by KiCS2. More details about this translation scheme can be found in [10,11].

As mentioned in the introduction, the KiCS2 implementation is based on the explicit representation of non-deterministic results in a data structure. This is achieved by extending each data type of the source program by constructors to represent a choice between two values and a failure, respectively. For instance, the data type for Boolean values defined in a Curry program by

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

is translated into the Haskell data type[1]

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

where `Fail` represents a failure and `(Choice i t1 t2)` a non-deterministic value, i.e., a selection of two values `t1` and `t2` that can be chosen by some search strategy. The first argument `i` of type `ID` of a `Choice` constructor is used to implement the call-time choice semantics discussed in Sect. 2. Since the evaluation of `xorSelf aBool` duplicates the argument operation `aBool`, we have to ensure that both duplicates, which later evaluate to a non-deterministic choice between two values, yield either `True` or `False`. This is obtained by assigning a unique identifier (of type `ID`) to each `Choice` constructor. The difficulty is to get a unique identifier on demand, i.e., when some operation evaluates to a `Choice`. We cannot thread an identifier supply, e.g., a counter, through the search tree without fixing an evaluation order. Since we want to compile into *purely* functional programs (in order to enable powerful program optimizations), we can neither use unsafe features with side effects to generate such identifiers. Hence, we follow the idea presented in [9] and pass a (conceptually infinite) set of identifiers, also called *identifier supply*, to each operation so that a `Choice` can pick its unique identifier from this set. For this purpose, we assume a type `IDSupply`, representing an infinite set of identifiers, with operations

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

The operation `initSupply` creates such a set (at the beginning of an execution), 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`. There are different implementations available (see below for a simple one) and our system is parametric over concrete implementations of `IDSupply`.

When translating Curry to Haskell, KiCS2 adds to each operation an additional argument of type `IDSupply`. For instance, the operation `aBool` defined in Sect. 2 is translated into:

[1] 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`.

```
aBool :: IDSupply → Bool
aBool s = Choice (thisID s) True False
```

Similarly, the operation

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

is translated into

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

so that the set `s` is split into a set `(leftSupply s)` containing identifiers for the evaluation of the argument `aBool` and a set `(rightSupply s)` containing identifiers for the evaluation of the operation `xorSelf`.

Since all data types are extended by additional constructors, we must also extend the definition of operations performing pattern matching.[2] For instance, consider the definition of polymorphic 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 then extended as described above:

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

The second rule transforms a non-deterministic argument into a non-deterministic result and the final rule returns `Fail` in all other cases, i.e., if `head` is applied to the empty list as well as if the matching argument is already a failed computation (failure propagation). Since deterministic operations do not introduce new `Choice` constructors, `head` does not use the identifier supply `s`.

To show a concrete example, we use the following implementation of `IDSupply` based on unbounded integers:

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

If we apply the same transformation to the rules defining `xor` and evaluate the main expression `(main 1)`, we obtain the result

[2] To obtain a simple compilation scheme, KiCS2 transforms source programs into uniform programs [11] where pattern matching is restricted to a single argument. This is always possible by introducing auxiliary operations.

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

Thus, the result is non-deterministic and contains three choices, whereby all of them have the same identifier. To extract all values from such a `Choice` structure, we have to traverse it and compute all possible choices but consider the choice identifiers to make consistent (left/right) decisions. 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 2 False True)` so that `False` is the only value possible for this branch. 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 2 True False)`, which again yields `False` as the only possible value. In consequence, the unintended value `True` is not extracted as a result.

The requirement to make consistent decisions can be implemented by storing the decisions already made for some choices during the traversal. For this purpose, we introduce the type

```
data Decision = NoDecision | ChooseLeft | ChooseRight
```

where `NoDecision` represents the fact that the value of a choice has not been decided yet. Furthermore, we assume operations to lookup the current decision for a given identifier or change it (depending on the implementation of `IDSupply`, KiCS2 supports several implementations based on memory cells or finite maps). For a top-level operation that prints all values contained in a choice structure in a depth-first manner, these operations would be of the following types:

```
lookupDecision :: ID → IO Decision
setDecision    :: ID → Decision → IO ()
```

Now the search operation can be defined by the I/O operation below:[3]

```
printValsDFS :: a → IO ()

printValsDFS Fail              = return ()

printValsDFS (Choice i x1 x2) = lookupDecision i >>= follow
  where
    follow ChooseLeft  = printValsDFS x1
    follow ChooseRight = printValsDFS x2
    follow NoDecision  = do newDecision ChooseLeft  x1
                            newDecision ChooseRight x2

    newDecision d x = do setDecision i d
                         printValsDFS x
                         setDecision i NoDecision

printValsDFS v = print v
```

This operation ignores failures and prints values that are not rooted by a `Choice` constructor. For a `Choice` constructor, it checks whether a decision for this identifier has already been made (note that the initial value for all identifiers is `NoDecision`). If a decision has been made for this choice, it follows this decision. Otherwise, the left alternative is used and this decision is stored. After printing all values w.r.t. this

[3] Note that this code has been simplified and slightly renamed compared to [11] for readability. The type system of Haskell does not allow this direct definition.

decision, the decision is undone (like in backtracking) and the right alternative is used and stored.

In general, this operation is applied to the normal form of the main expression (where `initSupply` is used to compute an initial identifier supply passed to this expression). The normal form computation is necessary for structured data, like lists, so that a failure or choice in some part of the data is moved to the root.

Other search strategies, like breadth-first search, iterative deepening, or parallel search, can be obtained by different implementations of this top-level operation to print all values. Instead of printing them, one can also collect the values in a tree-like data structure for further processing. Thus, KiCS2 supports a primitive

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

that returns the search tree corresponding to the evaluation of its argument, where the search tree is some computed value, a failure, or a choice between two trees:

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

This primitive is useful to encapsulate non-deterministic operations and select some result value, e.g., the "first" or the "best" one according to some ordering. Since the search tree is created in a demand-driven manner, the primitive is also applicable to infinite search spaces (in contrast to Prolog's `findall` primitive [25]). Based on this representation, a Curry programmer can define his own search strategies as tree traversals in his source program without any modification of the Curry compiler (see [19] for detailed examples). Note that these kinds of applications demand for a side-effect free implementation of non-deterministic computations (in contrast to traditional Prolog implementations [1])—which is the challenge addressed in this paper.

Since large parts of typical functional logic computations are deterministic, KiCS2 performs an optimization for deterministic operations. If an operation is defined by non-overlapping rules and does not call, neither directly nor indirectly through other operations, an operation defined by overlapping rules, the evaluation of such an operation (like `head`) cannot introduce non-deterministic values. Thus, it is not necessary to pass an identifier supply to the operation. In consequence, only the matching rules are extended by additional cases for handling `Choice` and `Fail` so that the generated code is nearly identical to a corresponding functional program. Actually, the benchmarks presented in [11] show that for deterministic operations this implementation outperforms all other Curry implementations, and, for non-deterministic operations, outperforms Prolog-based implementations of Curry and can compete with MCC [24], a Curry implementation that compiles to C.

As mentioned in the introduction, KiCS2 translates occurrences of logic variables into generators. For instance, the expression "`not x`", where `x` is a logic variable, is translated into "`not (aBool s)`", where `s` is an `IDSupply` provided by the context of the expression. The latter expression is evaluated by reducing the argument `(aBool s)` to a choice between `True` or `False` followed by applying `not` to this choice. This is similar to a narrowing step [28] on "`not x`" that instantiates the variable `x` to `True` or `False`. Since such generators are standard non-deterministic operations, they are translated like any other operation and, therefore, do not require any additional run-time support. However, in the presence of equational constraints, there are methods which are more efficient than generating all values. These methods and their implementation are discussed in the next section.

4 Equational Constraints and Unification

As known from logic programming, predicates or constraints are important to restrict the set of intended values in a non-deterministic computation. Apart from user-defined predicates, equational constraints of the form $e_1 =:= e_2$ are the most important kind of constraints. We have already seen a typical application of an equational constraint in the operation `last` in Sect. 2.

Due to the presence of non-terminating operations and infinite data structures, "$=:=$" is interpreted as the *strict equality* on terms [14], i.e., the equation $e_1 =:= e_2$ is satisfied iff e_1 and e_2 are reducible to unifiable constructor terms. In particular, expressions that do not have a value are not equal w.r.t. "$=:=$", e.g., the equational constraint "`head [] =:= head []`" is not satisfiable.[4]

According to this definition, "$=:=$" can be considered as a binary function defined by the following rules (we only present the rules for the Boolean and list types, where `Success` denotes the only constructor of the type `Success` of constraints):

```
True   =:= True    =  Success
False  =:= False   =  Success

[]     =:= []      =  Success
(x:xs) =:= (y:ys)  =  x =:= y & xs =:= ys

Success & c  =  c
```

If we translate these operations into Haskell by the scheme presented in Sect. 3, the following rules are added to these rules in order to propagate choices and failures:

```
Fail         =:= _             =  Fail
_            =:= Fail          =  Fail
Choice i l r =:= y             =  Choice i (l =:= y) (r =:= y)
x            =:= Choice i l r  =  Choice i (x =:= l) (x =:= r)
_            =:= _             =  Fail

Fail         & _     =  Fail
Choice i l r & c     =  Choice i (l & c) (r & c)
_            & _     =  Fail
```

Although this is a correct implementation of equational constraints, it might lead to an unnecessarily large search space when it is applied to generators representing logic variables. For instance, consider the following generator for Boolean lists:

```
aBoolList = [] ? (aBool : aBoolList)
```

This is translated into Haskell as follows:

[4] From now on, we use the standard notation for lists, i.e., `[]` denotes the empty list and `(x:xs)` denotes a list with head element `x` and tail `xs`.

```
aBoolList :: IDSupply → [Bool]
aBoolList s = Choice (thisID s) [] (aBool (leftSupply s)
                                     : aBoolList (rightSupply s))
```

Now consider the equational constraint "`x =:= [True]`". If the logic variable `x` is replaced by `aBoolList`, the translated expression "`aBoolList s =:= [True]`" creates a search space when evaluating its first argument, although there is no search required since there is only one binding for `x` satisfying the constraint. Furthermore and even worse, unifying two logic variables introduces an infinite search space. For instance, the expression "`xs =:= ys & xs++ys =:= [True]`" results in an infinite search space when the logic variables `xs` and `ys` are replaced by generators.

To avoid these problems, we have to implement the idea of the well-known unification principle [29]. Instead of enumerating all values for logic variables occurring in an equational constraint, we *bind* the variables to another variable or term. Since we compile into a purely functional language, the binding cannot be performed by some side effect. Instead, we add binding constraints to the computed results to be processed by a search strategy that extracts values from choice structures.

To implement unification, we have to distinguish free variables from "standard choices" (introduced by overlapping rules) in the target code. For this purpose, we refine the definition of the type `ID` as follows:[5]

```
data ID = ChoiceID Integer | FreeID Integer
```

The new constructor `FreeID` identifies a choice corresponding to a free variable, e.g., the generator for Boolean variables is redefined as

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

If an operation is applied to a free variable and requires its value, the free variable is transformed into a standard choice. For this purpose, we define a simple operation to perform this transformation:

```
narrow :: ID → ID
narrow (FreeID i) = ChoiceID i
narrow x          = x
```

We use this operation in narrowing steps, i.e., in all rules operating on `Choice` constructors. For instance, in the implementation of the operation `not` we replace the rule

```
not (Choice i x1 x2) s = Choice i (not x1 s) (not x2 s)
```

by the rule

```
not (Choice i x1 x2) s = Choice (narrow i) (not x1 s) (not x2 s)
```

to ensure that the resulting choice is not considered a free variable.

As mentioned above, the consideration of free variables is relevant in equational constraints where *binding constraints* are generated. For this purpose, we introduce a type to represent a binding constraint as a pair of a choice identifier and a decision for this identifier:

```
data Constraint = ID :=: Decision
```

[5] For the sake of simplicity, in the following, we consider the implementation of `IDSupply` to be unbounded integers.

Furthermore, we extend each data type by the possibility to add constraints:

```
data Bool   = ... | Guard [Constraint] Bool
data List a = ... | Guard [Constraint] (List a)
```

A single `Constraint` provides the decision for one constructor. In order to support constraints for structured data, a list of `Constraints` provides the decision for the outermost constructor and the decisions for all its arguments. Thus, (`Guard` *cs v*) represents a *constrained value*, i.e., the value *v* is only valid if the constraints *cs* are consistent with the decisions previously made during search. These binding constraints are created by the equational constraint operation "`=:=`": if a free variable should be bound to a constructor, we make the same decisions as it would be done in the successful branch of the generator. In case of Boolean values, this can be implemented by the following additional rules for "`=:=`":

```
Choice (FreeID i) _ _ =:= True  = Guard [i :=: ChooseLeft ] Success
Choice (FreeID i) _ _ =:= False = Guard [i :=: ChooseRight] Success
```

Hence, the binding of a variable to some known value is implemented as a binding constraint for the choice identifier for this variable. However, if we want to bind a variable to another variable, we cannot store a concrete decision. Instead, we store the information that the decisions for both variables, when they are made to extract values, must be identical. For this purpose, we extend the `Decision` type to cover this information:

```
data Decision = ... | BindTo ID
```

Furthermore, we add to the definition of "`=:=`" the rule that an equational constraint between two variables yields a binding for these variables:

```
Choice (FreeID i) _ _ =:= Choice (FreeID j) _ _
   =  Guard [i :=: BindTo j] Success
```

The consistency of constraints is checked when values are extracted from a choice structure, e.g., by the operation `printValsDFS`. For this purpose, we extend the definition of the corresponding search operations by calling a constraint solver for the constraints. For instance, the definition of `printValsDFS` is extended by a rule handling constrained values:

```
...
printValsDFS (Guard cs x) = do consistent <- add cs
                               if consistent then do printValsDFS x
                                                     remove cs
                                             else return ()
...
```

The operation `add` checks the consistency of the constraints `cs` with the decisions made so far and, in case of consistency, stores the decisions made by the constraints. In this case, the constrained value is evaluated before the constraints are removed to allow backtracking. Furthermore, the operations `lookupDecision` and `setDecision` are extended to deal with bindings between two variables, i.e., they follow variable chains in case of `BindTo` constructors.

Finally, with the ability to distinguish free variables (choices with an identifier of the form (`FreeID ...`)) from other values during search, values containing logic variables can also be printed in a specific form rather than enumerating all values, similarly to logic programming systems. For instance, KiCS2 evaluates the application of `head` to an unknown list as follows:

```
Prelude> head xs where xs free
{xs = (_x2:_x3)} _x2
```

Here, free variables are marked by the prefix `_x`.

5 Functional Patterns

A well-known disadvantage of equational constraints is the fact that "`=:=`" is interpreted as strict equality. Thus, if one uses equational constraints to express requirements on arguments, the resulting operations might be too strict. For instance, the equational constraint in the condition defining `last` (see Sect. 2) requires that `ys++[e]` as well as `xs` must be reducible to unifiable terms so that in consequence the input list `xs` is completely evaluated. Hence, if `failed` denotes an operation whose evaluation fails, the evaluation of `last [failed,True]` has no result. On the other hand, the evaluation of `last' [failed,True]` yields the value `True`, i.e., the definition of `last'` is less strict thanks to the use of functional patterns. Beyond this improved operational behavior, functional patterns can lead to more expressive programs (e.g., matching and unification on infinite structures, pattern matching at arbitrary depth in recursive data structures) and more elegant program patterns (see [4,7,17] for examples).

Conceptually, a functional pattern like `(xs++[e])` abbreviates all values to which it can be evaluated (by narrowing), like `[e]`, `[x1,e]`, `[x1,x2,e]`, and so on. In consequence, the rule defining `last'` abbreviates the following (infinite) set of rules:

```
last' [e] = e
last' [x1,e] = e
last' [x1,x2,e] = e
...
```

Obviously, one cannot implement functional patterns by a transformation into an infinite set of rules. Instead, they are implemented by a specific *lazy unification* procedure "`=:<=`" [4]. For instance, the definition of `last'` is transformed into

```
last' ys | (xs++[e]) =:<= ys  = e   where xs, e free
```

The behavior of "`=:<=`" is similar to "`=:=`", except for the case that a variable in the left argument should be bound to some expression: instead of evaluating the expression to some value and binding the variable to the value, the variable is bound to the *unevaluated* expression (see [4] for more details). Due to this slight change, failures or infinite structures in actual arguments do not cause problems in the matching of functional patterns.

Our proposed implementation of functional patterns in KiCS2 has a structure that is quite similar to that of equational constraints with the exception that variables could be also bound to unevaluated expressions. Only if such variables are later accessed, the expressions they are bound to are evaluated. This can be achieved by adding a further alternative to the type of decisions:

```
data Decision = ... | LazyBind [Constraint]
```

The implementation of the lazy unification operation "=:<=" is almost identical to the strict unification operation "=:=" as shown in Sect. 4. The only difference is in the rules where a free variable occurs in the left argument. All these rules are replaced by the single rule

```
Choice (FreeID i) _ _ =:<= x
  = Guard [i :=: LazyBind (lazyBind i x)] Success
```

where the auxiliary operation `lazyBind` implements the demand-driven evaluation of the right argument `x`:

```
lazyBind :: ID → a → [Constraint]
lazyBind i True  = [i :=: ChooseLeft]
lazyBind i False = [i :=: ChooseRight]
```

The use of the additional `LazyBind` constructor allows the argument `x` to be stored in a binding constraint without evaluation (due to the lazy evaluation strategy of the target language Haskell). However, it is evaluated by `lazyBind` to head normal form when its binding is required by another part of the computation, whereas the binding constraints for any sub-expression are in turn lazily computed using `lazyBind`.

Similarly to equational constraints, lazy bindings are processed by a solver when values are extracted. In particular, if a variable has more than one lazy binding constraint (which is possible if a functional pattern evaluates to a non-linear term), the corresponding expressions are evaluated and unified according to the semantics of functional patterns [4].

In order to demonstrate the operational behavior of our implementation, we sketch the evaluation of the lazy unification constraint `xs++[e] =:<= [failed,True]` that occurs when the expression `last' [failed,True]` is evaluated (we omit failed branches and some other details). Note that logic variables are replaced by generators, i.e., we assume that `xs` is replaced by `aBoolList 2` and `e` is replaced by `aBool 3`:

```
   aBoolList 2 ++ [aBool 3] =:<= [failed, True]
⇝ [aBool 4, aBool 3] =:<= [failed, True]
⇝ aBool 4 =:<= failed & aBool 3 =:<= True & [] =:<= []
⇝ Guard [ 4 :=: LazyBind (lazyBind 4 failed)
        , 3 :=: LazyBind (lazyBind 3 True)] Success
```

If the value of the expression `last' [failed,True]` is later required, the value of the variable `e` (with the identifier `3`) is in turn required. Thus, `(lazyBind 3 True)` is evaluated to `[3 :=: ChooseLeft]` which corresponds to the value `True` of the generator `(aBool 3)`. Note that the variable with identifier `4` does not occur anywhere else so that the binding `(lazyBind 4 failed)` will never be evaluated, as intended.

6 Benchmarks

In this section we evaluate our implementation of equational constraints and functional patterns by some benchmarks. The benchmarks were executed on a Linux machine running Debian 5.0.7 with an Intel Core 2 Duo (3.0GHz) processor. KiCS2 has been used with the Glasgow Haskell Compiler (GHC 7.0.4, option -O2) as its backend and an efficient `IDSupply` implementation that makes use of `IORefs`. For a comparison with other mature implementations of Curry, we considered PAKCS [18] (version

Expression	`==`	`=:=`	`=:<=`
`last (map (inc 0) [1..10000])`	2.91	0.05	0.01
`simplify`	10.30	6.77	7.07
`varInExp`	2.34	0.24	0.21
`fromPeano (half (toPeano 10000))`	26.67	5.95	11.19
`palindrome`	30.86	14.05	20.26
`horseman`	3.24	3.31	n/a
`grep`	1.06	0.10	n/a

Fig. 1. Benchmarks: comparing different representations for equations

1.9.2, based on a SICStus-Prolog 4.1.2) and MCC [24] (version 0.9.10). 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. The programs used for the benchmarks, partially taken from [4], are `last` (compute the last element of a list),[6] `simplify` (simplify a symbolic arithmetic expression), `varInExp` (non-deterministically return a variable occuring in a symbolic arithmetic expression), `half` (compute the half of a Peano number using logic variables), `palindrome` (check whether a list is a palindrome), `horseman` (solving an equation relating heads and feet of horses and men based on Peano numbers), and `grep` (string matching based on a non-deterministic specification of regular expressions [6]).

In Sect. 4 we mentioned that equational constraints could also be solved by generators without variable bindings, but this technique might increase the search space due to the possibly superfluous generation of all values. To show the beneficial effects of our implementation of equational constraints with variable bindings, in Fig. 1 we compare the results of using equational constraints (=:=) to the results where the Boolean equality operator (==) is used (which does not perform bindings but enumerate all values). As expected, in most cases the creation and traversal of a large search space introduced by "==" is much slower than our presented approach with variable bindings. In addition, the example `last` shows that the lazy unification operator (=:<=) improves the performance when unifying an expression which has to be evaluted only partially. Using strict unification, all elements of the list are (unnecessarily) evaluated. On the other hand, lazy unification causes some overhead when the expressions are fully evaluated, which is shown by the `fromPeano` and `palindrome` examples. Thus, it is reasonable to use it only if its improved computational power is really required, as intended by its design.

In contrast to the Curry implementations PAKCS and MCC, our implementation of strict unification is based on an explicit representation of the search space instead of backtracking and manipulating a global state containing bindings for logic variables. Nevertheless, the benchmarks in Fig. 2, using equational constraints only, show that it can compete with or even outperform the other implementations. The results show that the implementation of unification of MCC performs best. However, in most cases our implementation outperforms the Prolog-based PAKCS implementation, except for some examples. In particular, `simplify` does not perform well due to expensive

[6] "`inc x n`" is a naive addition that `n` times increases its argument `x` by 1.

Expression	KiCS2	PAKCS	MCC
`last (map (inc 0) [1..10000])`	0.05	0.40	0.01
`simplify`	6.77	0.15	0.00
`varInExp`	0.24	0.89	0.07
`fromPeano (half (toPeano 10000))`	5.95	108.88	3.22
`palindrome`	14.05	32.56	1.07
`horseman`	3.31	8.70	0.42
`grep`	0.10	2.88	0.14

Fig. 2. Benchmarks: strict unification in different Curry implementations

Expression	KiCS2	PAKCS
`last (map (inc 0) [1..10000])`	0.01	0.33
`simplify`	7.07	0.27
`varInExp`	0.21	1.87
`fromPeano (half (toPeano 10000))`	11.19	∞
`palindrome`	20.26	∞

Fig. 3. Benchmarks: functional patterns in different Curry implementations

bindings of free variables to large arithmetic expressions in unsuccessful branches of the search. Further investigation and optimization will hopefully lead to a better performance in such cases.

As MCC does not support functional patterns, the performance of lazy unification is compared with PAKCS only (Fig. 3). Again, our compiler performs well against PAKCS and outperforms it in most cases ("∞" denotes a run time of more than 30 minutes).

7 Conclusions and Related Work

We have presented an implementation of equational constraints and functional patterns in KiCS2, a purely functional implementation of Curry. In addition to the kernel implementation described in [11], we add binding constraints to computed values which are processed when values are extracted at the top level of a computation. Since only new constructors and pattern matching rules for them are added in our implementation, no overhead is introduced for programs without equational constraints, i.e., our implementation does not sacrifice the high efficiency of the kernel implementation shown in [11]. However, if these features are used, they usually lead to a comparably efficient execution, as demonstrated by our benchmarks. Although the benchmarks were small in order to evaluate our unification implementation, it should be noted that KiCS2 is used in larger applications, like the curricula and module information system of our department[7]. In these and similar applications, where large parts are purely functional computations, KiCS2 is 15-20 times faster than PAKCS [18].

Other implementations of equational constraints in functional logic languages are based on side effects. For instance, PAKCS [18] exploits the implementation of logic

[7] `http://www-ps.informatik.uni-kiel.de/~mh/studiengaenge/`

variables in Prolog, which are implemented on the primitive level by side effects. MCC [24] compiles into C where a specific abstract machine implements the handling of logic variables. We have shown that our implementation is competitive to those. In contrast to those systems, our implementation supports a variety of "top-level" search strategies, like iterative deepening, breadth-first or parallel search, as well as user-programmable search strategies, where the avoidance of side effects is important.

For future work it might be interesting to add further constraint structures to our implementation, like real arithmetic or finite domain constraints. This might be possible by extending the kinds of constraints of our implementation and solving them by functional programming approaches like [30].

References

1. Aït-Kaci, H.: Warren's Abstract Machine. MIT Press (1991)
2. Antoy, S., Hanus, M.: Compiling Multi-Paradigm Declarative Programs into Prolog. In: Kirchner, H. (ed.) FroCos 2000. LNCS (LNAI), vol. 1794, pp. 171–185. Springer, Heidelberg (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.: 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)
6. Antoy, S., Hanus, M.: Functional Logic Programming. Communications of the ACM 53(4), 74–85 (2010)
7. Antoy, S., Hanus, M.: New Functional Logic Design Patterns. In: Kuchen, H. (ed.) WFLP 2011. LNCS, vol. 6816, pp. 19–34. Springer, Heidelberg (2011)
8. 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)
9. Augustsson, L., Rittri, M., Synek, D.: On generating unique names. Journal of Functional Programming 4(1), 117–123 (1994)
10. Braßel, B., Fischer, S.: From Functional Logic Programs to Purely Functional Programs Preserving Laziness. In: Scholz, S.-B., Chitil, O. (eds.) IFL 2008. LNCS, vol. 5836, pp. 25–42. Springer, Heidelberg (2011)
11. Braßel, B., Hanus, M., Peemöller, B., Reck, F.: KiCS2: A New Compiler from Curry to Haskell. In: Kuchen, H. (ed.) WFLP 2011. LNCS, vol. 6816, pp. 1–18. Springer, Heidelberg (2011)
12. 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)
13. Braßel, B., Huch, F.: The Kiel Curry System KiCS. In: Seipel, D., Hanus, M., Wolf, A. (eds.) INAP/WLP 2007. LNCS (LNAI), vol. 5437, pp. 195–205. Springer, Heidelberg (2009)
14. Giovannetti, E., Levi, G., Moiso, C., Palamidessi, C.: Kernel LEAF: A Logic plus Functional Language. Journal of Computer and System Sciences 42(2), 139–185 (1991)
15. 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)

16. Hanus, M.: Multi-paradigm Declarative Languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
17. Hanus, M.: Declarative Processing of Semistructured Web Data. In: Technical Communications of the 27th International Conference on Logic Programming. Leibniz International Proceedings in Informatics (LIPIcs), vol. 11, pp. 198–208 (2011)
18. 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/`
19. Hanus, M., Peemöller, B., Reck, F.: Search Strategies for Functional Logic Programming. In: Proc. of the 5th Working Conference on Programming Languages (ATPS 2012), LNI, vol. 199, pp. 61–74. Springer (2012)
20. Hanus, M., Sadre, R.: An Abstract Machine for Curry and its Concurrent Implementation in Java. Journal of Functional and Logic Programming 1999(6) (1999)
21. Hanus, M. (ed.): Curry: An Integrated Functional Logic Language, Vers. 0.8.3 (2012), `http://www.curry-language.org`
22. Hussmann, H.: Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. Journal of Logic Programming 12, 237–255 (1992)
23. López-Fraguas, F., Sánchez-Hernández, J.: TOY: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
24. Lux, W.: Implementing Encapsulated Search for a Lazy Functional Logic Language. In: Middeldorp, A., Sato, T. (eds.) FLOPS 1999. LNCS, vol. 1722, pp. 100–113. Springer, Heidelberg (1999)
25. Naish, L.: All Solutions Predicates in Prolog. In: Proc. IEEE Internat. Symposium on Logic Programming, Boston, pp. 73–77 (1985)
26. O'Donnell, M.J.: Equational Logic as a Programming Language. MIT Press (1985)
27. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press (2003)
28. Reddy, U.S.: Narrowing as the Operational Semantics of Functional Languages. In: Proc. IEEE Internat. Symposium on Logic Programming, Boston, pp. 138–151 (1985)
29. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12(1), 23–41 (1965)
30. Schrijvers, T., Stuckey, P., Wadler, P.: Monadic Constraint Programming. Journal of Functional Programming 19(6), 663–697 (2009)
31. Slagle, J.R.: Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. Journal of the ACM 21(4), 622–642 (1974)

On the Efficient Implementation of Mode-Directed Tabling

João Santos and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
{jsantos,ricroc}@dcc.fc.up.pt

Abstract. Mode-directed tabling is an extension to the tabling technique that supports the definition of modes for specifying how answers are inserted into the table space. In this paper, we focus our discussion on the efficient support for mode-directed tabling in the YapTab tabling system, which uses *tries* to implement the table space. We discuss 7 different modes and explain how we have extended and optimized YapTab's table space organization to provide engine support for them. Experimental results, in the context of benchmarks taking advantage of mode-directed tabling, show that our implementation compares favorably with the B-Prolog and XSB state-of-the-art tabling systems.

1 Introduction

Tabling [1] is a recognized and powerful implementation technique that solves some limitations of Prolog's operational semantics in dealing with recursion and redundant sub-computations. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property*[1]. Tabling consists of saving and reusing the results of sub-computations during the execution of a program and, for that, the calls and the answers to tabled subgoals are stored in a proper data structure called the *table space*. The tabling technique can be viewed as a natural tool to implement dynamic programming algorithms. Dynamic programming is a general recursive strategy that consists in dividing a problem in simple sub-problems that, often, are really the same. Tabling is thus suitable to use with this kind of problems since, by storing and reusing intermediate results while the program is executing, it avoids performing the same computation several times.

In a traditional tabling system, all the arguments of a tabled subgoal call are considered when storing answers into the table space. When a new answer is not a variant[2] of any answer that is already in the table space, then it is always considered for insertion. Therefore, traditional tabling is very good for problems

[1] A logic program has the bounded term-size property if there is a function $f : N \rightarrow N$ such that whenever a query goal Q has no argument whose term size exceeds n, then no term in the derivation of Q has size greater than $f(n)$.

[2] Two terms are considered to be variant if they are the same up to variable renaming.

that require storing all answers. However, with dynamic programming, usually, the goal is to dynamically calculate optimal or selective answers as new results arrive. Writing dynamic programming algorithms can thus be a difficult task without further support. *Mode-directed tabling* [2] is an extension to the tabling technique that supports the definition of *modes* for specifying how answers are inserted into the table space. The idea is to use the modes to define the arguments to be considered for variant checking and to define how variant answers should be tabled regarding the remaining arguments. Mode-directed tabling has proved its viability for applications areas such as Machine Learning [3], Justification [4], Preferences [5], Answer Subsumption [6], among others.

To evaluate a predicate p/n using traditional tabling, we just need to declare it as '*table* p/n'. With mode-directed tabling, tabled predicates are declared using statements of the form '*table* $p(m_1, ..., m_n)$', where the m_i's are modes for the arguments. Implementations of mode-directed tabling are already available in ALS-Prolog [2] and B-Prolog [3], and a restricted form of mode-directed tabling can also be reproduced in XSB Prolog by using *answer subsumption* [7].

In this paper, we focus our discussion on the efficient implementation of mode-directed tabling in the YapTab tabling system [8], which uses *tries* [9] to implement the table space. Our implementation uses a more general approach to the declaration and use of modes and, currently, it supports 7 different modes: *index*, *first*, *last*, *min*, *max*, *sum* and *all*. To the best of our knowledge, no other tabling system supports all these modes and, in particular, the *sum* mode is not supported by any other system. Experimental results, using a set of benchmarks that take advantage of mode-directed tabling, show that our implementation compares favorably with the B-Prolog and XSB state-of-the-art tabling systems. This work is already fully integrated with the latest development version of Yap[3].

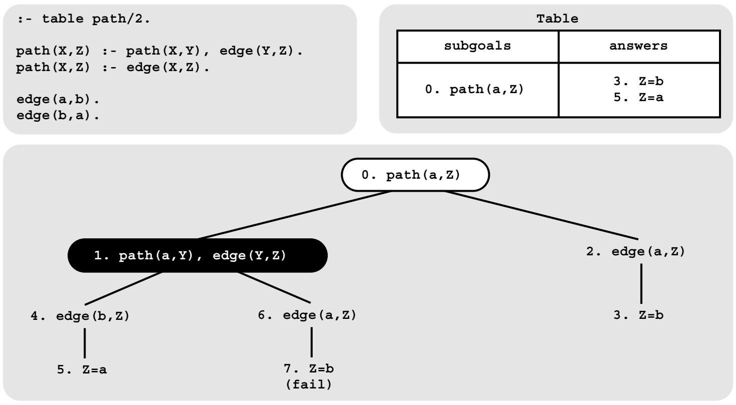
The remainder of the paper is organized as follows. First, we introduce some background concepts about tabling. Next, we describe the modes and we show examples of their use. Then, we introduce YapTab's table space organization and describe how we have extended it to efficiently support mode-directed tabling. At last, we present experimental results and outline some conclusions.

2 Tabled Evaluation

In tabling, variant calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in the corresponding table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all variant calls.

Figure 1 illustrates the execution of a tabled program. The top left corner shows the program code and the top right corner shows the final state of the table space. The program defines a small directed graph, represented by two *edge/2* facts, with a relation of reachability, defined by a *path/2* tabled predicate. The bottom of the figure shows the evaluation sequence, numbered in order of evaluation, for the query goal *path(a,Z)*.

[3] `http://www.dcc.fc.up.pt/~vsc/Yap`



```
:- table path/2.

path(X,Z) :- path(X,Y), edge(Y,Z).
path(X,Z) :- edge(X,Z).

edge(a,b).
edge(b,a).
```

Table

subgoals	answers
0. path(a,Z)	3. Z=b 5. Z=a

Fig. 1. An example of a tabled evaluation

First calls to tabled subgoals correspond to generator nodes (nodes depicted by white oval boxes) and, for first calls, a new entry, representing the subgoal, is added to the table space (step 0). Next, *path(a,Z)* is resolved against the first matching clause, calling in the continuation *path(a,Y)* (step 1). Since *path(a,Y)* is a variant call to *path(a,Z)*, we do not evaluate the subgoal against the program clauses, instead we consume answers from the table space. Such nodes are called *consumer nodes* (nodes depicted by black oval boxes). However, at this point, the table does not have answers for this call, so the computation is suspended[4].

The only possible move after suspending is to backtrack and try the second matching clause for *path(a,Z)* (step 2). This produces the answer $\{Z{=}b\}$, which is then stored in the table space (step 3). At this point, the computation at node 1 can be resumed with the newly found answer (step 4), giving rise to one more answer, $\{Z{=}a\}$ (step 5). This second answer is then also stored in the table space and propagated to the consumer node (step 6), which produces the answer $\{Z{=}b\}$ (step 7). This answer had already been found at step 3. Tabling does not store duplicate answers in the table space and, instead, variant answers *fail*. This is how tabling avoids unnecessary computations, and even looping in some cases. Since there are no more answers to consume nor more clauses left to try, the table entry for *path(a,Z)* is then marked as *completed*.

3 Mode-Directed Tabling

Traditional tabling can be viewed as a composition of two procedural operations: *Generate*() and *Aggregate*(). The *Generate*() operation corresponds to performing tabled evaluation from where a *bag of answers* is generated, i.e., we

[4] We are assuming a *suspension-based tabling* mechanism, where a tabled evaluation can be seen as a sequence of computations that suspend and later resume. Alternatively, *linear tabling* mechanisms use iterative computations to compute fix-points and for that they maintain a single execution tree (no suspension is needed).

may have duplicate (and infinite) answers as in Prolog. The *Aggregate*() operation then defines the criterion for specifying how answers are tabled which, for traditional tabling, is to eliminate variant answers.

Mode-directed tabling can be thought of as an extension to the *Aggregate*() operation that allows to define alternative criteria for specifying how variant answers (the index arguments) should be tabled (the output arguments). Index arguments are represented with mode *index*, while arguments with modes *first*, *last*, *min*, *max*, *sum* and *all* represent output arguments. Given the generic declaration $p(m_1, ..., m_j, m_{j+1}, ..., m_n)$, where for $1 <= i <= j$, m_i is an index argument and for $j+1 <= i <= n$, m_i is an output argument, the $Aggregate(p/n)$ operation can be defined as the set of answers:

$$\begin{aligned}\{p(x_1, ..., x_n) \mid \exists(z_{j+1}, ..., z_n) : p(x_1, ..., x_j, z_{j+1} ..., z_n) \in Generate(p/n) \\ \wedge\ x_{j+1} \in m_{j+1}(Out_{j+1}(x_1, ..., x_j)) \\ \wedge \ldots \\ \wedge\ x_n \in m_n(Out_n(x_1, ..., x_{n-1}))\} \\ \text{where } Out_j(x_1, ..., x_{j-1}) = \{y \mid \exists(z_{j+1}, ..., z_n) : \\ p(x_1, ..., x_{j-1}, y, z_{j+1} ..., z_n) \in Generate(p/n)\}\end{aligned}$$

For example, consider a $p/3$ predicate declared as $p(index, min, all)$ and the set of answers $\{p(a, 2, 2), p(b, 2, 1), p(b, 1, 2), p(b, 1, 1)\}$. The $Aggreg(p/3)$ operation is then:

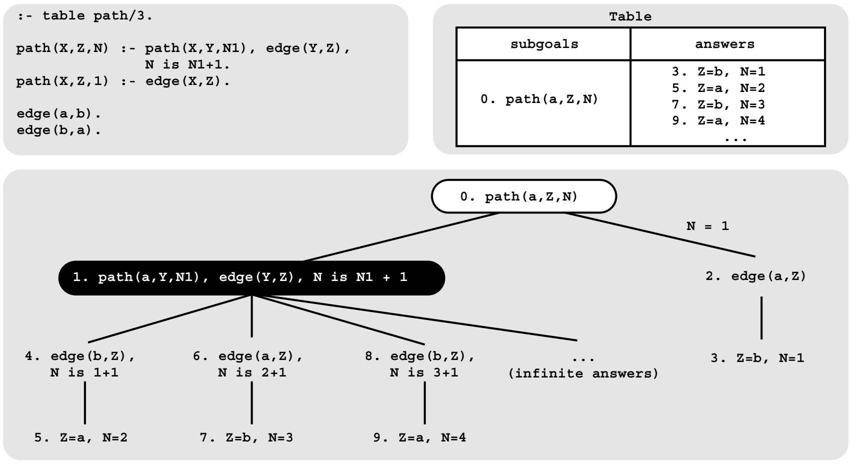
$$\begin{aligned}\{p(x_1, x_2, x_3) \mid\ \exists(z_2, z_3) : p(x_1, z_2, z_3) \in \{p(a, 2, 2), p(b, 2, 1), p(b, 1, 2), p(b, 1, 1)\} \\ \wedge\ x_2 \in min(Out_2(x_1)) \wedge\ x_3 \in all(Out_3(x_1, x_2))\}\end{aligned}$$

Since $min(Out_2(a)) = min(\{2\}) = \{2\}$, $all(Out_3(a, 2)) = max(\{2\}) = \{2\}$ and $min(Out_2(b)) = min(\{2, 1\}) = \{1\}$, $all(Out_3(b, 1)) = all(\{2, 1\}) = \{2, 1\}$ then $Aggreg(p/3) = \{p(a, 2, 2), p(b, 1, 2), p(b, 1, 1)\}$.

3.1 Index/First/Last Modes

Starting from the example in Fig. 1, consider now that we modify the program so that it also calculates the number of edges traversed in a path. As we can see in Fig. 2, the program does not terminate. Such behavior occurs because there is a path with an infinite number of edges starting from *a*, thus not satisfying the bounded term-size property necessary to ensure termination. In particular, the answers found at steps 3 and 7 and at steps 5 and 9 have the same answer for variable *Z* ({*Z=b*} and {*Z=a*}, respectively), but they are both inserted in the table space because they are not variants for variable *N*. For programs with an infinite number of answers, traditional tabling is thus not enough.

In Fig. 2, the problem relies on the fact that the third argument generates an infinite number of answers. We can thus define the *path/3* predicate as *path(index,index,first)* meaning that only the first and second arguments must be considered for variant checking and that, for the third argument, only the first answer must be tabled. With this declaration, the answer {*Z=b, N=3*} found at step 7 is no longer inserted in the table space and execution fails.



```
:- table path/3.

path(X,Z,N) :- path(X,Y,N1), edge(Y,Z),
               N is N1+1.
path(X,Z,1) :- edge(X,Z).

edge(a,b).
edge(b,a).
```

Table

subgoals	answers
0. path(a,Z,N)	3. Z=b, N=1 5. Z=a, N=2 7. Z=b, N=3 9. Z=a, N=4 ...

Fig. 2. A tabled evaluation with an infinite number of answers

The *last* mode implements the opposite behavior of the *first* mode, i.e., it always stores the last answer being found and discards the previous one, if any. Remember that with tabling, the order of answers is not important. However, in a particular implementation, the order of answers may depend on the tabling mechanism and on the evaluation strategy being use. Hence, we may question the necessity and/or correctness of the *first* and *last* modes.

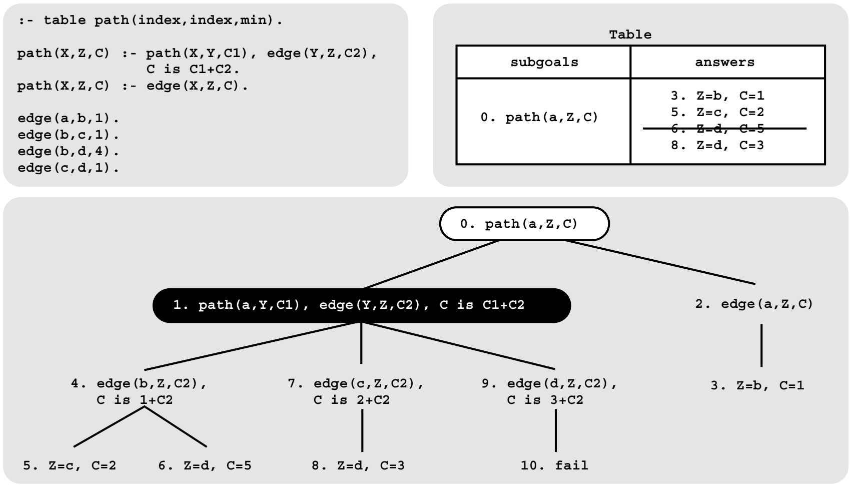
The *first* mode can be seen as a way to prune the search space, once an answer is found. This mode can also be read as *any*, *don't care* or *none*. We adopted the name *first* mainly to reflect the fact that, at the implementation level, we are storing the first answer as a way to represent a justification for that.

On the other hand, the *last* mode can be seen as a way to dynamically compute *preferable answers*. It is usually used in conjunction with a *preferable predicate* that is responsible for computing the preferable answers as new results arrive or fail if no preferable answer exists. In particular, all the other modes can be reproduced by using the *last* mode with appropriate preferable predicates. Please refer to [5,6] for examples where the *last* mode has shown to be very useful for implementing problems involving Preferences and Answer Subsumption.

3.2 Min/Max Modes

The *min* and *max* modes allow to specify a selective criterion that stores, respectively, the minimal and maximal answers for an argument. At the implementation level, we assume that when using the *min* and/or *max* modes, a tabled predicate is monotonic. Figure 3 shows an example using the *min* mode. The program's goal is to compute the paths with the shortest distances. The *path/3* predicate is declared as *path(index,index,min)*, meaning that the third argument should store only the minimal answers for the first two arguments.

By observing the example in Fig. 3, the most interesting part happens at step 8, where the answer $\{Z{=}d,\ C{=}3\}$ is found. This answer is a variant of the answer



```
:- table path(index,index,min).

path(X,Z,C) :- path(X,Y,C1), edge(Y,Z,C2),
               C is C1+C2.
path(X,Z,C) :- edge(X,Z,C).

edge(a,b,1).
edge(b,c,1).
edge(b,d,4).
edge(c,d,1).
```

Table

subgoals	answers
0. path(a,Z,C)	3. Z=b, C=1 5. Z=c, C=2 ~~6. Z=d, C=5~~ 8. Z=d, C=3

0. path(a,Z,C)
1. path(a,Y,C1), edge(Y,Z,C2), C is C1+C2
2. edge(a,Z,C)
4. edge(b,Z,C2), C is 1+C2
7. edge(c,Z,C2), C is 2+C2
9. edge(d,Z,C2), C is 3+C2
3. Z=b, C=1
5. Z=c, C=2
6. Z=d, C=5
8. Z=d, C=3
10. fail

Fig. 3. Using the *min* mode to compute the paths with the shortest distances

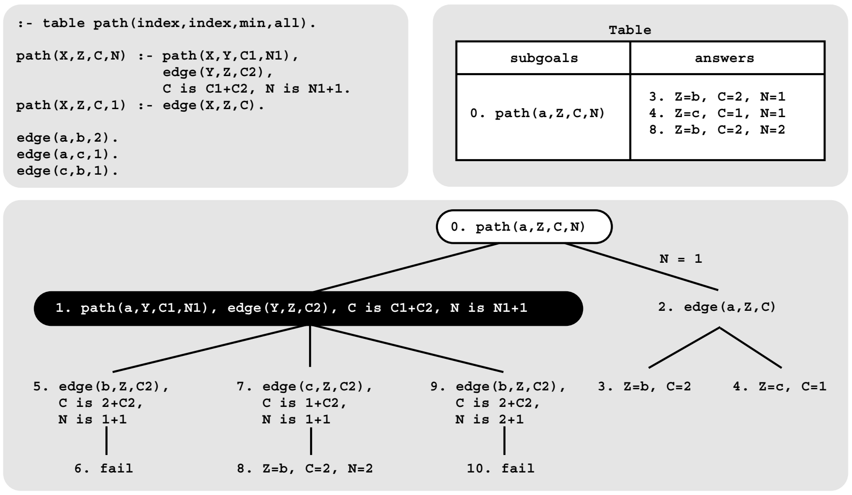
{*Z=d, C=5*} found at step 6. In the previous example, with the *first* mode, the old answer would have been kept in the table. Here, as the new answer is minimal on the third argument, the old answer is replaced by the new answer.

The *max* mode works similarly, but stores the maximal answer instead. For programs without the bounded term-size property, we must be careful when using these two modes as they may not ensure termination. For instance, this would be the case if, in Fig.3, we used the *max* mode instead of the *min* mode.

3.3 Sum/All Modes

Two other modes are the *sum* and the *all*. The *sum* mode allows to sum all the answers for an argument and the *all* mode allows to store all the answers. Consider now the example in Fig. 4 where a *path/4* predicate is declared as *path(index,index,min,all)* meaning that, for each path, we want to store the shortest distance (third argument) and, for the paths with the same shortest distances, the number of edges traversed (fourth argument). By following the example, the most interesting part happens when the answer {*Z=b, C=2, N=2*} is found at step 8. This answer is a variant of the answer found at step 3 and although both have the same minimal value (*C=2*), the new answer is still inserted in the table space since the number of edges (fourth argument) is different.

Notice that when the *sum* or *all* modes are used in conjunction with another mode, like the *min* mode in the example, it is important to keep in mind that the aggregation of answers made for the *sum* or *all* argument depends on the corresponding answer for the *min* argument. Consider, for example, that in the previous example we had found one more answer {*Z=b, C=1, N=4*}. In this case, the new answer would be inserted and the answers {*Z=b, C=2, N=1*} and {*Z=b, C=2, N=2*} would be deleted because the new answer corresponds to a shorter distance, as defined by the value *C=1* in the *min* argument.



```
:- table path(index,index,min,all).

path(X,Z,C,N) :- path(X,Y,C1,N1),
                 edge(Y,Z,C2),
                 C is C1+C2, N is N1+1.
path(X,Z,C,1) :- edge(X,Z,C).

edge(a,b,2).
edge(a,c,1).
edge(c,b,1).
```

Table

subgoals	answers
0. path(a,Z,C,N)	3. Z=b, C=2, N=1 4. Z=c, C=1, N=1 8. Z=b, C=2, N=2

0. path(a,Z,C,N)
N = 1
1. path(a,Y,C1,N1), edge(Y,Z,C2), C is C1+C2, N is N1+1
2. edge(a,Z,C)
5. edge(b,Z,C2), C is 2+C2, N is 1+1
7. edge(c,Z,C2), C is 1+C2, N is 1+1
9. edge(b,Z,C2), C is 2+C2, N is 2+1
3. Z=b, C=2
4. Z=c, C=1
6. fail
8. Z=b, C=2, N=2
10. fail

Fig. 4. Using the *all* mode to compute the paths with the shortest distances together with the number of edges traversed

3.4 Related Work

The ALS-Prolog [2] and B-Prolog [3] systems also implement mode-directed tabling but using a different syntax. For example, the *index* and *first* modes are known as + and - and in ALS-Prolog the *all* mode is known as @. The *sum* mode is not supported by any other system and B-Prolog also does not implement the *last* and *all* modes. On the other hand, B-Prolog includes an extra mode, named *nt*, to indicate that a given argument should not be tabled and, thus, not considered to be inserted in the table space. B-Prolog also extends the mode-directed tabling declaration to include a *cardinality limit* that allows to define the maximum number of answers to be stored in the table space [3].

Mode-directed tabling can also be reproduced in the XSB system by using two *answer subsumption* mechanisms [7]. One is called *partial order answer subsumption* and can be used to mimic, in terms of functionality, the *min* and *max* modes. Consider that we want to use it with the program in Fig. 3 that computes the paths with the shortest distances. Then, we should declare the *path/3* predicate as $path(\_,\_,po(</2))$ meaning that the third argument will be evaluated using partial order answer subsumption, where $</2$ implements the partial order relation. The other two arguments are considered to be index arguments.

The other XSB's mechanism, called *lattice answer subsumption*, is more powerful and can be used to mimic, in terms of functionality, the other modes. Considering the same example, we only need to change the *path/3* declaration to $path(\_,\_,lattice(min/3))$. The *min/3* predicate has three arguments since, with this mechanism, we must generate a third answer starting from the new answer and from the answer stored in the table:

$$min(Old, New, Res) :- \ Old < New \rightarrow Res = Old \ ; \ Res = New.$$

4 Implementation

In this section, we start by presenting some background about the table space organization in YapTab and then we discuss in more detail how we have extended it to efficiently support mode-directed tabling.

4.1 YapTab's Table Space Organization

Like we have seen, during the execution of a program, the table space may be accessed in a number of ways: (i) to find out if a subgoal is in the table and, if not, insert it; (ii) to verify whether a newly or preferable answer is already in the table and, if not, insert it; and (iii) to load answers from the tables.

With these requirements, a careful design of the table space is critical to achieve an efficient implementation. YapTab uses *tries* which is regarded as a very efficient way to implement the table space [9]. A trie is a tree structure where each different path through the *trie nodes* corresponds to a term described by the tokens labeling the traversed nodes. For example, the tokenized form of the term $path(X, 1, f(Y))$ is the sequence of 5 tokens $path/3$, VAR_0, 1, $f/1$ and VAR_1, where each variable is represented as a distinct VAR_i constant. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second term $path(Z, 1, b)$. Since the main functor, token $path/3$, and the first two arguments, tokens VAR_0 and 1, are common to both terms, only one additional node will be required to fully represent this second term in the trie, thus allowing to save three nodes in this case.

YapTab implements tables using two levels of tries. The first level, named *subgoal trie*, stores the tabled subgoal calls and the second level, named *answer trie*, stores the answers for a given call. More specifically, each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate's subgoal trie. Each different subgoal call is then represented as a unique path in the subgoal trie, starting at the table entry and ending in a *subgoal frame* data structure, with the argument terms being stored within the path's nodes. The subgoal frame data structure acts as an entry point to the answer trie. Contrary to subgoal tries, answer trie paths hold just the substitution terms for the free variables that exist in the argument terms of the corresponding call [9].

An example for a tabled predicate $p/3$ is shown in Fig. 5. Initially, the table entry for $p/3$ points to an empty subgoal trie. Then, the subgoal $p(X, 1, Y)$ is called and three trie nodes are inserted to represent the arguments in the call: one for variable X (VAR_0), a second for integer 1, and a last one for variable Y (VAR_1). Since the predicate's functor term is already represented by its table entry, we can avoid inserting an explicit node for $p/3$ in the subgoal trie. Then, the leaf node is set to point to a subgoal frame, from where the answers for the call will be stored. The example shows two answers for $p(X, 1, Y)$: $\{X{=}VAR_0,\ Y{=}f(VAR_1)\}$ and $\{X{=}VAR_0,\ Y{=}b\}$. Since both answers have the same substitution term for argument X, they share the top node in the answer trie (VAR_0). For argument Y, each answer has a different substitution term and, thus, a different path is used to represent each.

When adding answers, the leaf nodes are chained in a linked list in insertion time order, so that the recovery may happen the same way. In Fig. 5, we can observe that the leaf node for the first answer (node VAR_1) points (dashed arrow) to the leaf node of the second answer (node b). To maintain this list, two fields in the subgoal frame data structure point, respectively, to the first and last answer of this list (for simplicity of illustration, these pointers are not shown in Fig. 5). When consuming answers, a consumer node only needs to keep a pointer to the leaf node of its last loaded answer, and consumes more answers just by following the chain. Answers are loaded by traversing the trie nodes bottom-up (again, for simplicity of illustration, such pointers are not shown in Fig. 5).

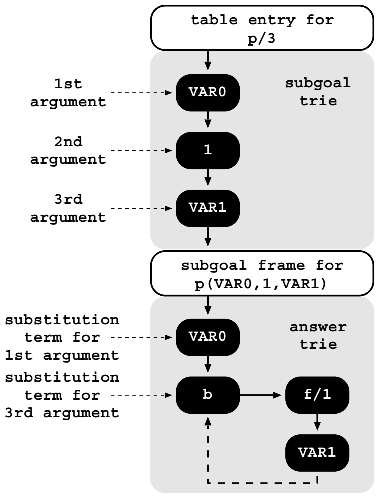


table entry for p/3
1st argument
VAR0
subgoal trie
2nd argument
1
3rd argument
VAR1
subgoal frame for p(VAR0,1,VAR1)
substitution term for 1st argument
VAR0
answer trie
substitution term for 3rd argument
b
f/1
VAR1

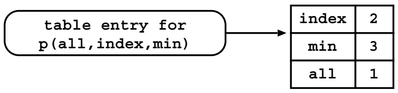
Fig. 5. Table space organization

4.2 Mode-Directed Tabled Subgoal Calls

In YapTab, mode-directed tabled predicates are compiled by extending the table entry data structure to include a *mode array*, where the information about the modes is stored. In this mode array, the modes appear in the order in which the arguments are accessed, which can be different from their position in the original declaration. For example, *index* arguments must be considered first, irrespective of their position. Or, if using the *all* and *min* modes in a declaration, all *min* arguments must be considered before any *all* argument, since the *all* means that all answers must be stored, making meaningless the notion of being minimal in this case. As we will see in Section 4.3, changing the order is also strictly necessary to achieve an efficient implementation. In YapTab, the mode information is thus stored in the order mentioned below, together with the argument's position:

1. arguments with *index* mode;
2. arguments with *min* and *max* mode;
3. arguments with *all* mode;
4. arguments with *last* or *first* or *sum* (only one *sum* argument is allowed) mode (the combination of different modes is not allowed).

Figure 6 shows an example for a *p(all,index,min)* mode-directed tabled predicate. The *index* mode is placed first in the mode array, then the *min* mode and last the *all* mode. With traditional tabling, tabled calls are inserted in their own subgoal tries by following the order of the arguments in the call. With mode-directed tabling, we follow the order defined in



index	2
min	3
all	1

Fig. 6. Mode array

the corresponding mode array. Figure 7 shows the difference between the resulting subgoal tries with and without mode-directed tabling for the subgoal call *p(X,1,Y)*. The values in the mode array indicate that we should start by inserting first the second argument of the subgoal call (1), then the third argument (Y or VAR_0) and last the first argument (X or VAR_1).

The mode information is used when creating the subgoal frame associated with the subgoal call at hand. With mode-directed tabling, subgoal frames were extended to include a new array, named *substitution array*, where the mode information is stored, together with the number of free variables associated with each argument in the subgoal call. The argument's order is the same as in the mode array. Figure 8 shows the substitution array for the subgoal call *p(X,1,Y)*. The first position, corresponding to the argument with constant 1, has no free variables and thus we store a 0 in the substitution array. The other two arguments are free variables and, thus, they have a 1 in the substitution array. It is possible to optimize the array by removing entries that have 0 variables and by joining contiguous entries with the same mode. As we will see next, the substitution array plays an important role in the process of inserting answers in the answer trie.

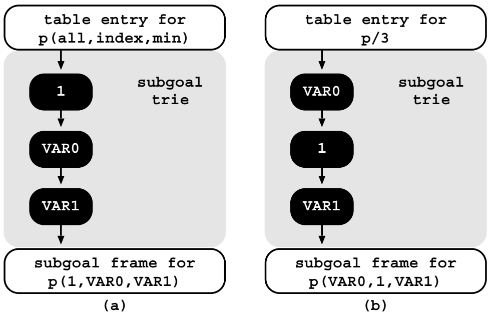
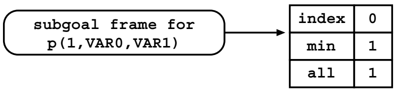


table entry for p(all,index,min)
subgoal trie
1
VAR0
VAR1
subgoal frame for p(1,VAR0,VAR1)
(a)
table entry for p/3
subgoal trie
VAR0
1
VAR1
subgoal frame for p(VAR0,1,VAR1)
(b)

Fig. 7. Subgoal tries for *p(X,1,Y)* considering *p/3* declared (a) with and (b) without mode-directed tabling

4.3 Mode-Directed Tabled Answers

Like in traditional tabling, tabled answers are only represented by the substitution terms for the free variables in the arguments of the corresponding subgoal call. However, for mode-directed tabling, when we are considering the substitution terms individually, it is important to know beforehand which mode applies to each, and for that, we use the information stored in the corresponding substitution array.



index	0
min	1
all	1

Fig. 8. Substitution array

Consider again the substitution array for the subgoal call *p(X,1,Y)*. Now, if we find the answer $\{X{=}f(a),\ Y{=}5\}$, the first binding to be considered is $\{Y{=}5\}$ with *min* mode and then $\{X{=}f(a)\}$ with *all* mode. Please note that the substitutions are considered in the same order that the variables they substitute have been inserted in the subgoal trie. Since the answer trie is initially empty, both terms can be inserted as usual. Later, if another answer is found, for example, $\{X{=}b,\ Y{=}3\}$, we begin the insertion process by considering the binding $\{Y{=}3\}$ with *min* mode. As there is already an answer in the table, we must compare both accordingly to the *min* mode. Since the new answer is preferable ($3 < 5$), the old answer must be *invalidated* and the new one inserted in the table. The invalidation process consists in: (a) deleting all intermediate nodes corresponding to the answers being invalidated; and (b) tagging the leaf nodes of such answers as *invalid nodes*. Invalid nodes are only deleted when the table is later completed or abolished. Figure 9 illustrates the aspect of the answer trie before and after the invalidation process.

Invalid nodes are opaque to subsequent subgoal calls, but can be still visible from the consumer calls already in evaluation. Hence, when invalidating a node, we may have consumers still pointing to it. By deleting leaf nodes, this would make consumers unable to follow the chain of answers. An alternative would be to traverse the stacks and update the consumers pointing to invalidated answers, but this could be a very costly operation.

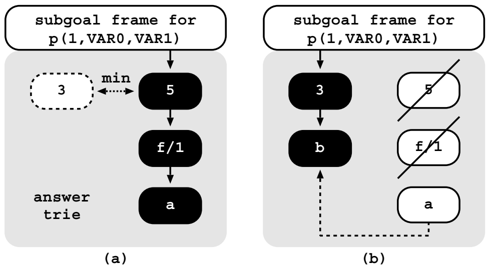

subgoal frame for p(1,VAR0,VAR1)
3
min
5
f/1
answer trie
a
(a)
subgoal frame for p(1,VAR0,VAR1)
3
5
b
f/1
a
(b)

Fig. 9. Aspect of the answer trie (a) before and (b) after the invalidation process

Notice also that the mode's order in the substitution array is crucial for the simplicity and efficiency of the invalidation process. When, at a given node N, we decide that an answer should be invalidated, the substitution array's order ensures that all nodes below node N (including N) are the ones we want to invalidate and that the upper nodes are the ones we want to keep.

This might not be the case if we used a *bad* order. For example, Fig. 10 illustrates the aspect of the answer trie before the invalidation process if we considered the original arguments order for *p(X,1,Y)*. In Fig. 10, to detect that the second answer is preferable ($3 < 5$), we need to navigate in the trie until reaching the leaf node 5 for the first answer. Thus, the invalidation process may require deleting upper nodes (as the example in Fig. 10 shows) and/or traverse several paths to fully detect all preferable answers (this would be the case if we had two intermediate answers with the same minimal values, for instance $\{X{=}f(a),\ Y{=}5\}$ and $\{X{=}h(c),\ Y{=}5\}$), making therefore the invalidation process much more complex and costly.

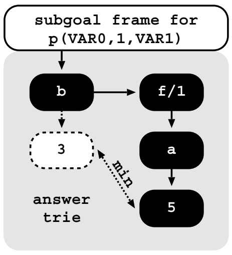


Fig. 10. Before the invalidation process if using a *bad* order

4.4 Scheduling and Mode-Directed Tabling

In a tabled evaluation, there are several points where we may have to choose between continuing forward execution, backtracking, consuming answers or completing subgoals. Such decision is determined by the scheduling strategy. The two most successful strategies are *batched scheduling* and *local scheduling* [10].

Batched scheduling evaluates programs in a depth-first manner as does the WAM. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues with forward execution. Only when all clauses have been resolved, the newly tabled answers will be forwarded to the consumers. Batched scheduling thus tries to delay the need to move around the search tree by batching the consumption of answers.

Local scheduling is an alternative strategy that tries to complete subgoals sooner. The key idea is that whenever new answers are added to the table space, the execution then fails. Local scheduling thus explores the whole search space for a tabled predicate before returning answers for forward execution.

To the best of our knowledge, YapTab is the only tabling system that supports the dynamic mixed-strategy evaluation of batched and local scheduling within the same evaluation [10]. This is very important, because for mode-directed tabled predicates, the ability of being able to use local evaluation can be crucial to correctly and/or efficiently support some modes.

This is the case for the *sum* mode, that we discuss next in more detail. As it sums all the answers for a given argument, we might end with wrong results if we return partial results instead of aggregating them and only returning the aggregated result. Consider, for example, the two mode-directed tabled predicates $num\_links/2$ and $num\_nodes/1$ in Fig. 11 and the query goal $num\_nodes(N)$. If $num\_links/2$ is evaluated using local scheduling, we get the right result (*N=3*) but, with batched scheduling, we end with a wrong result (*N=6*). This occurs because, with batched evaluation, the $num\_links(\_,\_)$ call in the second clause of $num\_nodes/2$ succeeds 2 times for each $edge/2$ fact. Moreover, with batched scheduling, there is no means to return the partial sums while the table is being computed. With local scheduling, since the result is only returned at the end, this problem does not apply.

$:-\ table\ num\_links(index, sum).$
$num\_links(A, 0) :-\ edge(\_, A).$
$num\_links(A, 1) :-\ edge(A, \_).$

$:-\ table\ num\_nodes(sum).$
$num\_nodes(0).$
$num\_nodes(1) :-\ num\_links(\_, \_).$

$edge(a, b). \qquad edge(a, c). \qquad edge(b, c).$

Fig. 11. A cascade of two mode-directed tabled predicates using the *sum* mode

Batched evaluation can also yield useless computations for mode-directed tabled predicates. Consider a $p(max)$ tabled predicate and the query goal:

$:-\ p(Max),\ do\_work(Max, Res).$

With batched evaluation, the call to $do\_work(Max, Res)$ will be executed for each Max partial result computed by $p(Max)$, hence producing many useless computations as the number of non-maximal results.

5 Experimental Results

In this section, we present some experimental results for a set of benchmarks that take advantage of mode-directed tabling. The environment for our experiment was a machine with a AMD FX(tm)-8150 8-core processor with 32 GBytes of main memory and running the Linux kernel 64 bits version 3.2.0. To put our results in perspective, we compare our implementation, on top of Yap Prolog (development version 6.3), with the B-Prolog (version 7.8 beta-6) and the XSB (version 3.3.6) systems, both using local scheduling. For XSB, we adapted the benchmarks to use lattice answer subsumption (as discussed in Section 3.4)[5]. For benchmarking, we used the following set of programs:

short(N) uses the *min* mode to determine all-pairs shortest paths in a graph representing the flight connections between the N busiest commercial airports in US[6].
short_first(N) uses the *first* mode to extend the all-pairs shortest paths program to also include the first justification for each shortest path.
short_all(N) uses the *all* mode to extend the all-pairs shortest paths program to also include all the justifications for each shortest path.
short_pref(N) uses the *last* mode to solve the all-pairs shortest paths program using Preferences [6].
knapsack(N) uses the *max* mode to determine the maximum number of items to include in a collection, from N weighted items, so that the total weight is equal to a given value.
lcs(N) uses the *max* mode to find the longest subsequence common to two different sequences of size N.
matrix(N) uses the *min* mode to implement the matrix chain multiplication problem that determines the most efficient way to multiply a sequence of N matrices.
pagerank(N) uses the *sum* mode to measure the rank values of web pages in a realistic dataset of web links called *search engines*[7], using N iterations.

Table 1 shows the execution times, in milliseconds, for running the benchmarks with YapTab, B-Prolog and XSB. In parentheses, it also shows the execution time ratios against YapTab with local evaluation. The execution times are the average of 3 runs. The entries marked with *n.a.* correspond to programs using modes not available in B-Prolog. The ratios marked with (—) mean that we are *not considering* them in the average results (they correspond either to *n.a.* entries or to execution times much higher than YapTab).

In addition to these results, we also collected some statistics for YapTab when running with local and batched evaluation. Table 2 shows the number of answer trie nodes (column **#nodes**) and the number of tabled answers (column **#ans**) present in the table space for YapTab at the end of the execution (columns **Final**) and the respective differences for the full execution with local and batched evaluation (columns **Extra/Deleted**). These differences represent the extra trie nodes and answers that were allocated/found during the evaluation and later deleted and, thus, are not present in the final tables.

[5] For programs using *min*/*max* modes, we also tried with partial order answer subsumption but, unexpectedly, we got worse results.
[6] `http://toreopsahl.com/datasets`
[7] `http://www.cs.toronto.edu/~tsap/experiments/download/download.html`

Table 1. Execution times, in milliseconds, for YapTab, B-Prolog and XSB and the respective ratios when compared with YapTab's local evaluation

Programs	**YapTab** Local	**YapTab** Batched	**B-Prolog**	**XSB**
short(300)	1,088	1,261 (1.16)	2,990 (2.75)	2,922 (2.69)
short(400)	1,544	1,785 (1.16)	4,216 (2.73)	4,321 (2.80)
short(500)	2,170	2,472 (1.14)	5,792 (2.67)	6,218 (2.87)
short_first(300)	1,394	2,641 (1.89)	3,225 (2.31)	5,013 (3.60)
short_first(400)	2,052	3,432 (1.67)	4,614 (2.25)	7,257 (3.54)
short_first(500)	2,866	4,488 (1.57)	6,379 (2.23)	10,328 (3.60)
short_all(300)	4,324	8,383 (1.94)	*n.a.* (—)	61,803 (—)
short_all(400)	5,861	10,590 (1.81)	*n.a.* (—)	122,985 (—)
short_all(500)	8,337	13,598 (1.63)	*n.a.* (—)	239,451 (—)
short_pref(300)	2,882	4,241 (1.47)	*n.a.* (—)	6,666 (2.31)
short_pref(400)	4,152	5,621 (1.35)	*n.a.* (—)	9,932 (2.39)
short_pref(500)	5,773	7,473 (1.29)	*n.a.* (—)	14,129 (2.45)
knapsack(1000)	1,013	998 (0.99)	837 (0.83)	2,684 (2.65)
knapsack(1500)	1,581	1,561 (0.99)	1,229 (0.78)	3,977 (2.52)
knapsack(2000)	2,037	2,040 (1.00)	1,582 (0.78)	5,473 (2.69)
lcs(1000)	1,196	1,170 (0.98)	2,900 (2.42)	3,060 (2.56)
lcs(1500)	2,768	2,722 (0.98)	5,784 (2.09)	7,128 (2.58)
lcs(2000)	4,864	4,804 (0.99)	10,116 (2.08)	13,338 (2.74)
matrix(100)	192	224 (1.17)	582 (3.03)	396 (2.06)
matrix(150)	925	1,076 (1.16)	2,549 (2.76)	1,610 (1.74)
matrix(200)	3,005	3,534 (1.18)	7,816 (2.60)	4,688 (1.56)
pagerank(1)	365	*n.a.* (—)	*n.a.* (—)	128,377 (—)
pagerank(16)	813	*n.a.* (—)	*n.a.* (—)	> 10 *min* (—)
pagerank(36)	1,260	*n.a.* (—)	*n.a.* (—)	> 10 *min* (—)
Average ratio		(1.31)	(2.15)	(2.63)

In general, the results show that, for all combinations of experiments and systems, there is no clear tendency showing that the execution time ratios increase or decrease as we increase the size of the corresponding set of programs.

Comparing the results for local and batched evaluation, they show that, on average, batched evaluation is around 31% worse than local evaluation. These results are confirmed in Table 2, where we can observe that batched evaluation always allocates/deletes more trie nodes and inserts/deletes more tabled answers than local evaluation. In particular, batched evaluation gets worse the more answers are inserted into the table space. This affects in particular the **short_first()**, **short_all()** and **short_pref()** set of programs, which confirms our discussion regarding the fact that, in general, local evaluation is more suitable to reduce the search space for mode-directed tabled predicates.

Regarding the comparison with the other systems, YapTab's results clearly outperform those of B-Prolog and XSB. On average, B-Prolog and XSB are, respectively, around 2.15 and 2.63 times worse than YapTab using local evaluation. Please note that for B-Prolog and XSB we do not include the performance of some programs into the average results. For B-Prolog, this is because these

Table 2. Number of answer trie nodes and tabled answers for YapTab at the end of the execution and the respective differences (extra nodes and answers allocated/found that were later deleted) for the full execution with local and batched evaluation

Programs	**Final**		**Extra/Deleted**			
			Local		**Batched**	
	#nodes	**#ans**	**#nodes**	**#ans**	**#nodes**	**#ans**
short(300)	179,401	89,401	77,911	77,911	586,488	586,488
short(400)	317,618	157,618	122,435	122,435	706,060	706,060
short(500)	500,000	250,000	196,831	196,831	877,913	877,913
short_first(300)	661,458	89,401	586,348	77,911	16,476,991	586,488
short_first(400)	1,213,352	157,618	947,584	122,435	18,733,939	706,060
short_first(500)	1,997,262	250,000	1,609,053	196,831	21,760,014	877,913
short_all(300)	2,615,740	690,614	5,609,890	1,584,000	30,418,627	4,740,397
short_all(400)	4,351,566	1,084,942	8,129,237	2,172,438	35,762,267	5,632,706
short_all(500)	6,806,102	1,611,082	12,039,458	3,017,929	43,281,969	6,835,251
short_pref(300)	179,401	89,401	77,911	77,911	586,488	586,488
short_pref(400)	317,618	157,618	122,435	122,435	706,060	706,060
short_pref(500)	500,000	250,000	196,831	196,831	877,913	877,913
knapsack(1000)	1,960,131	973,453	87,816	87,816	307,055	307,055
knapsack(1500)	2,963,665	1,475,220	109,613	109,613	450,276	450,276
knapsack(2000)	3,960,969	1,973,872	127,957	127,957	584,980	584,980
lcs(1000)	1,980,191	989,118	101,997	101,997	206,485	206,485
lcs(1500)	4,445,865	2,221,466	234,713	234,713	484,700	484,700
lcs(2000)	7,917,402	3,956,741	420,051	420,051	866,027	866,027
matrix(100)	10,100	5,050	11,089	11,089	14,862	14,862
matrix(150)	22,648	11,324	17,791	17,791	39,775	39,775
matrix(200)	40,194	20,097	36,325	36,325	68,848	68,848
pagerank(1)	85,111	30,896	1,825,175	1,240,703	*n.a.*	*n.a.*
pagerank(16)	378,783	104,314	3,237,305	1,711,413	*n.a.*	*n.a.*
pagerank(36)	741,343	194,954	4,828,085	2,241,673	*n.a.*	*n.a.*

programs use the *all*, *last* and *sum* modes, which are not supported in B-Prolog. For XSB, the execution times for the **short_all()** and **pagerank()** are much higher than YapTab and including them would have distorted the comparison between the three systems. To the best of our knowledge, YapTab is thus the only system that supports the *all*, *last* and *sum* modes and handles them efficiently.

6 Conclusions

We discussed how we have extended YapTab's table space organization to provide engine support for mode-directed tabling. In particular, we presented how we deal with mode-directed tabled subgoal calls and answers and we discussed the role of scheduling in mode-directed tabled evaluations. Our implementation uses a more general approach to the declaration and use of modes and, currently, it supports 7 different modes. To the best of our knowledge, no other tabling system supports all these modes and, in particular, the *sum* mode is not supported by any other system.

Experimental results on benchmarks that take advantage of mode-directed tabling, showed that our implementation clearly outperforms the B-Prolog and XSB state-of-the-art tabling systems. In particular, YapTab is the only system that efficiently handles programs that use the *all* mode.

Further work will include extending our implementation to support multi-threaded mode-directed tabling and explore the impact of applying mode-directed tabling to other problems.

Acknowledgments. This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects LEAP (FCOMP-01-0124-FEDER-015008) and HORUS (FCOMP-01-0124-FEDER-010074). João Santos is funded by the FCT grant SFRH/BD/76307/2011.

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43(1), 20–74 (1996)
2. Guo, H.F., Gupta, G.: Simplifying Dynamic Programming via Mode-directed Tabling. Software Practice and Experience 38(1), 75–94 (2008)
3. Zhou, N.F., Kameya, Y., Sato, T.: Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In: IEEE International Conference on Tools with Artificial Intelligence, vol. 2, pp. 213–218. IEEE Computer Society (2010)
4. Pemmasani, G., Guo, H.F., Dong, Y., Ramakrishnan, C.R., Ramakrishnan, I.V.: Online Justification for Tabled Logic Programs. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 24–38. Springer, Heidelberg (2004)
5. Guo, H.F., Jayaraman, B., Gupta, G., Liu, M.: Optimization with Mode-Directed Preferences. In: 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pp. 242–251. ACM (2005)
6. Santos, J., Rocha, R.: Mode-Directed Tabling and Applications in the YapTab System. In: Symposium on Languages, Applications and Technologies, pp. 25–40 (2012)
7. Swift, T., Warren, D.S.: Tabling with Answer Subsumption: Implementation, Applications and Performance. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS (LNAI), vol. 6341, pp. 300–312. Springer, Heidelberg (2010)
8. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming 5(1&2), 161–205 (2005)
9. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming 38(1), 31–54 (1999)
10. Rocha, R., Silva, F., Santos Costa, V.: Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 250–264. Springer, Heidelberg (2005)

The Generalized Intensional Transformation for Implementing Lazy Functional Languages[⋆]

Georgios Fourtounis[1], Nikolaos Papaspyrou[1], and Panos Rondogiannis[2]

[1] School of Electrical and Computer Engineering
National Technical University of Athens, Greece
[2] Department of Informatics and Telecommunications
University of Athens, Greece

Abstract. The intensional transformation is a promising technique for implementing lazy functional languages based on a demand-driven execution model. Despite its theoretical elegance and its simple and efficient execution model, the intensional transformation suffered, until now, from two main drawbacks: it could only be applied to programs that manipulate primitive data-types and it could only compile a simple (and rather restricted) class of higher-order functions. In this paper we remedy the above two deficiencies, obtaining a transformation algorithm that is applicable to mainstream lazy functional languages. The proposed transformation initially uses defunctionalization in order to eliminate higher-order functions from the source program. The original intensional transformation is then extended in order to apply to the target first-order language with user-defined data types that resulted from the defunctionalization. It is demonstrated that the proposed technique can be used to compile a relatively large subset of Haskell into portable C code whose performance is comparable to existing mainstream implementations.

Keywords: intensional transformation, dataflow programming, defunctionalization, compilation, lazy functional languages.

1 Introduction

The *intensional transformation* [20,16,17] has been proposed as an alternative technique for implementing lazy functional languages based on a demand-driven execution model. The key idea behind the intensional approach is to transform a source functional program into a program consisting of nullary variable definitions enriched with intensional (i.e., context-switching) operators. The transformation was initially proposed as a technique for implementing first-order functional languages [20] and was also used in the implementation of the first-order dataflow language Lucid [19]. Later on, the correctness of the transformation was formally established [16] and it was extended to apply to a simple

[⋆] Work partially supported by the research project "ΘΑΛΗΣ−ΕΜΠ: Handling uncertainty in data intensive applications on a distributed computing environment (cloud computing)" (MIS 380153), funded by the European Social Fund and the Greek national funds through the Operational Program "Education and Lifelong Learning".

class of higher-order programs [17], in which partially applied objects can only be top-level function names. For the class of programs that it can compile, the transformation has been demonstrated to be quite efficient [4].

Despite its theoretical elegance and its simple and efficient execution model, the intensional transformation continues to suffer from the two main drawbacks that were present since its inception:

- It can only be applied to programs with primitive data-types (such as integers, characters, boolean values, and so on). For example, the dataflow language Lucid never supported user-defined data-types [19, Sec. 7.1].
- It can only compile a simple (and rather restricted) class of higher-order functions. More specifically, the extension of the intensional transformation [17] can only compile programs that make a Pascal-like use of higher-order functions (i.e., programs that do not use function closures and therefore do not support currying in its full-generality).

In this paper we remedy the above two deficiencies, obtaining a transformation algorithm that is applicable to mainstream higher-order lazy functional languages. The proposed transformation initially uses defunctionalization [15] in order to eliminate higher-order functions from the source program (at the cost of introducing data constructors representing explicit closures in the target first-order program). In this way, the two problems above are trivially reduced to the first one. The first problem is then solved by demonstrating that the original intensional transformation can be appropriately extended to handle a language with user-defined data types (and pattern matching). This problem is solved in this paper, which extends an idea that was presented last year in an informal symposium [6]. It is also demonstrated that the proposed technique can be used to compile a relatively large subset of Haskell into portable C code whose performance is comparable to existing Haskell implementations, based on more traditional compilation techniques.

The rest of the paper is organized as follows: Section 2 provides background on the original intensional transformation and introduces the proposed generalized transformation at an intuitive level, whereas Section 3 presents a formalization thereof. Section 4 discusses the details of an implementation of the proposed technique. Section 5 provides a performance comparison with several well-known and efficient Haskell compilers. The paper concludes (Sections 6 and 7) with a discussion of related work and directions for future research.

2 From the Original to the Generalized Transformation

In this section we introduce the intensional transformation in an intuitive way. We start by outlining the original transformation (for an extensive discussion, see [20,16]) and proceed by sketching our new approach with a simple example.

2.1 The Original Intensional Transformation

The input to the original intensional transformation [20,16] is a first-order functional program that only uses base data-types (such as integers, Boolean values, and so on). We assume that all the variables in the program (i.e., function names and their formal parameters) are distinct; this can obviously be achieved by a straightforward preprocessing. The source program is then transformed into a zero-order *intensional* program that only contains nullary definitions. The name "intensional" reflects the fact that the resulting program additionally uses two context-switching operators, whose semantics will be shortly described. The transformation can be intuitively described as follows [16]:

1. Let `f` be a function defined in the source functional program. Number the textual occurrences of calls to `f` in the program, starting at 0 (including calls in the body of the definition of `f`).
2. Replace the i-th call of `f` in the program by $\texttt{call}_i\texttt{(f)}$. Remove the formal parameters from the definition of `f`, so that `f` is defined as an ordinary individual variable.
3. Introduce a new definition for each formal parameter of `f`. The right hand side of the definition is the operator `actuals` applied to a list of the actual parameters corresponding to the formal parameter in question, listed in the order in which the calls are numbered.

To illustrate the algorithm, consider the following simple first-order program on the left. The transformation produces the target program on the right:

```
result = f 3 + f 5              result = call0(f) + call1(f)
f x    = g (x*x)                f      = call0(g)
g y    = y+2                    g      = y+2
                                x      = actuals(3, 5)
                                y      = actuals(x*x)
```

The above intensional code can be easily evaluated with respect to an initially empty *context*. Evaluation contexts are in fact lists of natural numbers which, intuitively, keep track of the exact position in the recursion tree where the execution currently is. The operators $\texttt{call}_i$ and `actuals` are context-switching operators: $\texttt{call}_i$ augments a list w by prefixing it with i, whereas `actuals` takes the head i of a list, and uses it to select its i-th argument. One can now easily define an *EVAL* function which evaluates the intensional program that results from the transformation, as shown in Figure 1. The function is parameterized by the program p in which all evaluation takes place; this will often be omitted to simplify presentation. The function $body(v, p)$ returns the defining expression of a variable v in program p. The evaluation of the usual constructs of functional languages (if-then-else, arithmetic operations, etc.) are all expressed by the rule for n-ary constants `c` (which, when $n = 0$ also covers the case of nullary constants, such as numbers, characters, and so on). Notice that the order of evaluation in this case depends on the meaning of the constant `c`: if `c` is an arithmetic operator (e.g., "+") then the recursive calls to *EVAL* will have to

$$\begin{array}{lcl}
EVAL_p(v, w) & = & EVAL_p(body(v,p), w) \\
EVAL_p(\texttt{call}_i(e), w) & = & EVAL_p(e, i : w) \\
EVAL_p(\texttt{actuals}(e_0, \ldots, e_{n-1}), i : w) & = & EVAL_p(e_i, w) \\
EVAL_p(\texttt{c}(e_0, \ldots, e_{n-1}), w) & = & c(EVAL_p(e_0, w), \ldots, EVAL_p(e_{n-1}, w))
\end{array}$$

Fig. 1. The *EVAL* function for the intensional language

$$\begin{array}{l}
\quad EVAL(\texttt{result}, [\,]) \\
= EVAL(\texttt{call}_0\texttt{(f) + call}_1\texttt{(f)}, [\,]) \\
= EVAL(\texttt{call}_0\texttt{(f)}, [\,]) + EVAL(\texttt{call}_1\texttt{(f)}, [\,]) \\
= EVAL(\texttt{f}, [0]) + EVAL(\texttt{f}, [1]) \\
= EVAL(\texttt{call}_0\texttt{(g)}, [0]) + EVAL(\texttt{call}_0\texttt{(g)}, [1]) \\
= EVAL(\texttt{g}, [0,0]) + EVAL(\texttt{g}, [0,1]) \\
= EVAL(\texttt{y}, [0,0]) + EVAL(\texttt{2}, [0,0]) + EVAL(\texttt{y}, [0,1]) + EVAL(\texttt{2}, [0,1]) \\
= EVAL(\texttt{actuals(x*x)}, [0,0]) + 2 + EVAL(\texttt{actuals(x*x)}, [0,1]) + 2 \\
= EVAL(\texttt{x*x}, [0]) + 2 + EVAL(\texttt{x*x}, [1]) + 2 \\
= EVAL(\texttt{x}, [0]) * EVAL(\texttt{x}, [0]) + 2 + EVAL(\texttt{x}, [1]) * EVAL(\texttt{x}, [1]) + 2 \\
= EVAL(\texttt{actuals(3, 5)}, [0]) * EVAL(\texttt{actuals(3, 5)}, [0]) + 2 + \\
\quad EVAL(\texttt{actuals(3, 5)}, [1]) * EVAL(\texttt{actuals(3, 5)}, [1]) + 2 \\
= EVAL(\texttt{3}, [\,]) * EVAL(\texttt{3}, [\,]) + 2 + EVAL(\texttt{5}, [\,]) * EVAL(\texttt{5}, [\,]) + 2 \\
= 9 + 2 + 25 + 2 \;=\; 38
\end{array}$$

Fig. 2. Execution of the target intensional program

be computed strictly; if on the other hand `c` corresponds to a non-strict operator (e.g., if-then-else), then evaluation is dictated by the meaning of this operator.

The execution of our example intensional program derived above is given in Figure 2. Notice that we assume that all source programs have a distinguished variable `result` whose value we want to compute.

The evaluation function just described roughly corresponds to call-by-name: notice how `x` is evaluated again and again under the same context. To obtain a call-by-need implementation, one can use an appropriate warehouse, in which triples of the form (*variable*, *context*, *value*) are stored — see [16, Sec. 12] for a more extensive discussion on the history and details of this issue. Every time the value of a variable under a given context is demanded, the warehouse is searched. If an entry is found, the corresponding value is returned; otherwise, the value of the variable under the current context is computed and placed in the warehouse for possible future reuse. A more efficient way of memoizing results, using *lazy activation records* (LARs), has been proposed in [4]; the idea of LARs is generalized and used in Section 4.

2.2 The New Intensional Transformation

As mentioned in the introductory section, the intensional transformation was never generalized to apply to a fully higher-order functional language nor to a language that supports user-defined data-structures. From an implementation point of view, higher-order functions and data-structures are closely connected, since, using Reynold's defunctionalization, one can reduce a higher-order program to a first-order one that is enriched with appropriate data-structures [15]. In other words, the two problems can be simultaneously solved if we generalize the intensional transformation to apply to first-order programs with user-defined data types. For example, consider the following second-order Haskell program:

```
result  = inc (add 1) 2 + inc sq 3
inc f x = f (x+1)
add a b = a+b
sq z    = z*z
```

The source program is initially defunctionalized as shown below:

```
result     = inc (fadd 1) 2 + inc fsq 3
inc f x    = apply f (x+1)
add a b    = a+b
sq z       = z*z

data Func  = Fadd Int | Fsq
fadd c     = Fadd c
fsq        = Fsq

apply cl d = case cl of
               Fadd c → add c d
               Fsq    → sq d
```

The above is a standard defunctionalization with two small tricks. First, we have introduced functions `fadd` and `fsq` which have replaced all occurrences of the constructors `Fadd` and `Fsq`. Second, in the `case` pattern corresponding to `Fadd`, we have used the same variable `c` that appears in the definition of `fadd`. These two conventions (to be discussed more generally in Section 3) ensure that we can apply the intensional transformation and obtain an equivalent zero-order intensional program, exactly as we did before:

```
result = call_0(inc) + call_1(inc)
inc    = call_0(apply)
add    = a+b
sq     = z*z

fadd   = Fadd
fsq    = Fsq

apply  = case cl of
           Fadd → call_0(add)
           Fsq  → call_0(sq)

f      = actuals(call_0(fadd), fsq)
```

```
x      = actuals(2, 3)
a      = actuals(c)
b      = actuals(d)
z      = actuals(d)
c      = actuals(1)
cl     = actuals(f)
d      = actuals(x+1)
```

The above program can be executed following the same basic principles as the one presented in the previous subsection, using a demand-driven interpreter in the form of a function $EVAL_p(e, w)$ that will be defined formally in Section 3.

3 A Formal Account of the Generalized Transformation

In this section we present the generalized transformation in a more formal way. Since defunctionalization is a well-known and broadly used technique, in the following we will not discuss it any further. Instead, from now on we will assume that our source language is a lazy first-order functional language with user defined data-types (i.e., a language whose syntax matches the syntax of the programs that are produced by defunctionalization). We will call this language FOFL (First-Order Functional Language).

The syntax of FOFL is defined by the following context-free grammar, where f and v range over variables, c ranges over constants, κ ranges over constructors, and $n, m \geq 0$. When $n = 0$, we will omit the empty parentheses.

$$\begin{array}{rcll}
p & ::= & d_0,\ \ldots,\ d_n & \textit{program} \\
d & ::= & f(v_0,\ \ldots,\ v_{n-1}) = e & \textit{definition} \\
e & ::= & c(e_0,\ \ldots,\ e_{n-1}) \mid f(e_0,\ \ldots,\ e_{n-1}) \mid \kappa(e_0,\ \ldots,\ e_{n-1}) & \textit{expression} \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ \{\ b_0\ ;\ \ldots\ ;\ b_n\ \} \mid \texttt{\#}^m(v) & \\
b & ::= & \kappa(v_0,\ \ldots,\ v_{n-1}) \rightarrow e & \textit{case clause}
\end{array}$$

As outlined in the previous section, we assume that FOFL programs are in a *normalized form.* We assume that the formal parameters of all functions are distinct. This can be achieved by simple renaming. Furthermore, for each constructor κ with n arguments, there will be a function defined as:

$$f_\kappa(v_0, \ldots, v_{n-1}) = \kappa(v_0, \ldots, v_{n-1})$$

and all occurrences of κ in the program will be replaced by occurrences of f_κ. We also assume that patterns corresponding to κ in all `case` expressions will use the same variables $v_0, \ldots, v_{n-1}$ that appear in the definition of f_κ. Unfortunately, this cannot be achieved by simple renaming, as there may be nested `case` expressions. For this reason, we introduce a special form of expressions $\texttt{\#}^m(v)$ that will resolve such scoping issues.

Roughly speaking, $\texttt{\#}^m(v)$ corresponds to the variable v that is bound in a pattern of the m-th enclosing `case` expression. For example, function `apply` in the example of the previous section will be written as:

$$\begin{aligned} apply(cl,d) \;=\; & \texttt{case}\ cl\ \texttt{of}\ \{ \\ & \quad Add(c) \to add(\texttt{\#}^0(c), d); \\ & \quad Sq \to sq(d) \\ & \} \end{aligned}$$

where $\texttt{\#}^0(c)$ corresponds to the variable c bound by the pattern $Add(c)$ of the `case` expression. An example with nested `case` follows, where the expression on the left (in Haskell syntax, calculating the sum of the first two elements of a list) can be normalized as shown on the right:

```
case l of
  Nil → 0
  Cons x xs →
    case xs of
      Nil → x
      Cons y ys → x+y
```

```
case $l$ of {
    $Nil \to 0$;
    $Cons(h,t) \to$
        case $\#^0(t)$ of {
            $Nil \to \#^1(h)$;
            $Cons(h,t) \to +(\#^1(h), \#^0(h))$
        }
}
```

Notice here that the same set of variables (h, t) is used in both patterns for *Cons* and that `x` and `y`, which both correspond to h, are distinguished by the value of m (the nesting depth of `case` expressions).

3.1 The Generalized NVIL

FOFL programs are transformed into zero-order intensional ones in the language NVIL (Nullary Variables Intensional Language). For more background on such languages, the interested reader can consult the first sections of [16]. The only difference of NVIL from the corresponding language defined in [16] is that the former supports user-defined data types. The syntax of NVIL is given by the following context-free grammar. Notice that the syntax of the intensional operators (`call` and `actuals`) is slightly different from the one informally introduced in Section 2 and that $\texttt{\#}^m(v)$ has been replaced by the more general $\texttt{\#}^m(e)$.

$$\begin{array}{rcll} p & ::= & d_0,\ \ldots,\ d_n & \textit{program} \\ d & ::= & f = e & \textit{definition} \\ e & ::= & c(e_0,\ \ldots,\ e_{n-1}) \mid f \mid \kappa \mid \texttt{case}\ e\ \texttt{of}\ \{\ b_0\ ;\ldots\ ;\ b_n\ \} & \textit{expression} \\ & \mid & \texttt{\#}^m(e) \mid \texttt{call}_\ell(e) \mid \texttt{actuals}(\langle e_\ell \rangle_{\ell \in I}) & \\ b & ::= & \kappa \to e & \textit{case clause} \end{array}$$

In Section 2, operator `call` was labeled by a natural number i and operator `actuals` received a sequence of expressions, indexed by i. Here, we slightly change this and take the index to be any element ℓ from an appropriate set *Labels*. Therefore, `call` is labeled by ℓ and `actuals` receives a sequence of expressions e_ℓ indexed by labels ranging over a subset $I \subseteq \textit{Labels}$. We represent this sequence as $\langle e_\ell \rangle_{\ell \in I}$. This convention does not affect the semantics of NVIL

$$
\begin{array}{l}
EVAL_p(c(e_0,\ \ldots,\ e_{n-1}), w) \ = \ c(EVAL_p(e_0, w), \ldots, EVAL_p(e_{n-1}, w)) \\
EVAL_p(f, w) \ = \ EVAL_p(body(f, p), w) \\
EVAL_p(\kappa, w) \ = \ \langle \kappa, w \rangle \\
EVAL_p(\texttt{case}\ e\ \texttt{of}\ \{\kappa_0 \to e_0;\ \ldots;\ \kappa_n \to e_n\}, \langle \ell, w, \mu \rangle) \ = \ EVAL_p(e_i, \langle \ell, w, w' : \mu \rangle) \\
\quad \text{if } EVAL_p(e, \langle \ell, w, \mu \rangle) = \langle \kappa_i, w' \rangle \\
EVAL_p(\texttt{\#}^m(e), \langle \ell, w, \mu \rangle) \ = \ EVAL_p(e, \mu_m) \\
EVAL_p(\texttt{call}_\ell(e), w) \ = \ EVAL_p(e, \langle \ell, w, \bullet \rangle) \\
EVAL_p(\texttt{actuals}(\langle e_\ell \rangle_{\ell \in I}), \langle \ell, w, \mu \rangle) \ = \ EVAL_p(e_\ell, w)
\end{array}
$$

Fig. 3. Semantics of NVIL

but will be useful in the definition of the intensional transformation and its proof of correctness (not discussed in this paper).

The semantics of NVIL is given in Figure 3. As discussed in Section 2, it is defined in the form of an evaluation function $EVAL_p(e, w)$, where p is the program, e is the expression to be evaluated, and w is the intensional context. In contrast to the simple structure of contexts (lists of labels) used in [16], the introduction of user-defined data types requires a more complex kind of contexts, similar to lists with backpointers (b-lists) defined by Yaghi [20].

Contexts are defined by the following grammar. The new element is μ, which is a list of contexts corresponding to nested `case` expressions.

$$
\begin{array}{lcl}
w & ::= & \bullet \mid \langle \ell, w, \mu \rangle \\
\mu & ::= & \bullet \mid w : \mu
\end{array}
$$

The result of function $EVAL_p(e, w)$ is either a ground value, which is returned by the meaning of some operator c (e.g., an integer number), or a pair of the form $\langle \kappa, w \rangle$, which corresponds to a value of a user-defined data type. In the latter case, κ is the constructor that was used to build this value and w is the context that must be used to evaluate the constructor's arguments. This semantics is captured in the equation for $EVAL_p(\kappa, w)$; remember that such expressions can only occur in the bodies of functions f_κ that have been introduced for all constructors κ.

The semantics of `call` and `actuals` operate on the context in the same way as informally introduced in Section 2; `call` adds a new label to the context and `actuals` selects the expression to evaluate based on the current label, which it removes from the context. The most interesting parts of the semantics are the equations for `case` and for `#`^{m}. In the former, the expression to be analyzed is evaluated and is found to be of the form $\langle \kappa_i, w' \rangle$ for some constructor κ_i that is mentioned in one of the clauses of `case`. (This is guaranteed if the program is well typed and `case` clauses are exhaustive, but we do not discuss typing issues in this paper.) Evaluation proceeds with the body e_i of that clause but the context w' is prepended to the list μ of contexts corresponding to nested `case` expressions. If later, in the evaluation of e_i, an expression of the form `#`$^m(e)$ is found, the context μ_m found in the m-th position of the list μ is used for evaluating e, instead of the current context.

$$
\begin{array}{lcl}
\mathcal{E}(c(e_0,\ldots,e_{n-1})) & = & c(\mathcal{E}(e_0),\ldots,\mathcal{E}(e_{n-1})) \\
\mathcal{E}(f) & = & f \\
\mathcal{E}(f(e_0,\ldots,e_n)) & = & \texttt{call}_\ell(f) \quad \text{where} \;\; \ell = \langle e_0,\ldots,e_n \rangle \\
\mathcal{E}(\kappa(e_0,\ldots,e_{n-1})) & = & \kappa \\
\mathcal{E}(\texttt{case}\; e \;\texttt{of}\; \{b_0\texttt{;}\ldots\texttt{;}b_n\}) & = & \texttt{case}\; \mathcal{E}(e) \;\texttt{of}\; \{\mathcal{B}(b_0);\; \ldots;\; \mathcal{B}(b_n)\} \\
\mathcal{E}(\texttt{\#}^m(e)) & = & \texttt{\#}^m(\mathcal{E}(e)) \\
\mathcal{B}(\kappa(v_0,\;\ldots,\;v_{n-1}) \to e) & = & \kappa \to \mathcal{E}(e)
\end{array}
$$

$$
\mathit{actdefs}(f,p) \;=\; \bigcup_{j=0}^{n-1} \; \{v_j = \texttt{actuals}(\langle \mathcal{E}(l_j) \rangle_{l \in I})\}
$$

where $v_0,\ldots,v_{n-1}$ are the formal parameters of f and $I = \mathit{labels}(f,p)$

$$
\mathit{Trans}(p) \;=\; \bigcup_{f(v_0,\ldots,v_{n-1})=e \;\text{in}\; p} \; \{f = \mathcal{E}(e)\} \;\cup\; \mathit{actdefs}(f,p)
$$

Fig. 4. The transformation algorithm from FOFL to NVIL

3.2 The Intensional Transformation from FOFL to NVIL

We start by defining the set *labels*(f,p), i.e., the set of labels of calls to f in program p. These labels will form the indices of `call` operators. More specifically, the label of a function call $f(e_0,\ldots,e_{n-1})$ is simply the sequence of its arguments $\langle e_0,\ldots,e_{n-1}\rangle$. In other words, the transformed form of the call $f(e_0,\ldots,e_{n-1})$ will be $\texttt{call}_\ell$ where $\ell = \langle e_0,\ldots,e_{n-1}\rangle$. This assumption is slightly different from the one presented in Section 2.1 but it helps us in two ways. First, using this assumption, two identical function calls in the program receive exactly the same label. Second, since a label ℓ is a sequence of the actual parameters of a function call, we can write ℓ_m in order to specify the m-th actual parameter of this call. This helps us simplify notation. Recapitulating:

$$
\mathit{labels}(f,p) \;=\; \{\langle e_0,\ldots,e_{n-1}\rangle \mid f(e_0,\ldots,e_{n-1}) \text{ in } p\}
$$

We can now define the overall transformation from FOFL to NVIL, as shown in Figure 4. Given a program p, the function *Trans*(p) removes the formal parameters from all definitions and adds one extra definition for every formal parameter of every function in the program. The creation of these extra definitions is performed by the function *actdefs*. More specifically, given a function f with formal parameters $v_0,\ldots,v_{n-1}$, the function *actdefs*(f,p) creates one `actuals` definition for each v_j; this definition contains a sequence of all the (processed) actual parameters of f in p that correspond to the j-th position. Finally, we have the functions $\mathcal{E}$ and $\mathcal{B}$, which process expressions and `case` clauses. The main role of these two functions is to replace function calls with corresponding occurrences of the operator `call`.

4 The Implementation

In this section we describe an implementation of the generalized intensional transformation. The key idea of the implementation is that for every definition in the target intensional program, a corresponding piece of C code is generated, parameterized by the current context. In fact, the C code implements a more efficient version of the *EVAL* function in Figure 3. The runtime system uses a stack and a heap. However, in contrast to the standard implementation of user-defined data types that are represented as heap objects, the only entities that are stored in the stack and the heap are *Lazy Activation Records* (LARs), which we adapt here from our previous work [4]. A LAR is created when an expression of the form $\texttt{call}_\ell(f)$ is encountered during the execution of the program. LARs are similar to traditional activation records where, among other things, function parameters are stored. Some of the fields in a LAR are not filled at the time of the function call, when the LAR is constructed, but only when their value is actually demanded by the implementation. Notice that when the value of a formal parameter under a given context is demanded *again* during execution, then the existing value for this formal parameter can be retrieved from the LAR. In other words, the LARs implement a call-by-need semantics, as discussed at the end of Subsection 2.1.

A LAR corresponds directly to a context of the form $w = \langle \ell, w', \mu \rangle$ in the definition of function *EVAL* in Figure 3. More specifically, it contains the fields:

– *prev*: a pointer to the parent LAR, i.e., the LAR of the function that invoked this one. It corresponds directly to w' above.
– $arg_0, \ldots, arg_{n-1}$: each arg_i points to the code corresponding to the i-th formal parameter of the function call that generated this LAR. This is an encoding of ℓ, in the formal semantics of NVIL, and can be directly used to evaluate the function's arguments.
– $val_0, \ldots, val_{n-1}$: each val_i memoizes the value of the corresponding arg_i. It is initially empty and will be filled on demand: if at some point the code stored in arg_i is executed and computes a value, this value will be stored in val_i for future use. This implements a call-by-need semantics.
– *nested*: this field corresponds directly to μ. It is in fact an array which memoizes the values of expressions used in nested `case` constructs. In particular, when an expression of the form $\texttt{\#}^m(e)$ is later encountered, *nested*$[m]$ points to the LAR that must be used to evaluate e.

With all this in mind, the compilation of the NVIL program to C code faithfully follows the rules of $EVAL_p$ given in Figure 3.

The main difference between our approach and the standard implementation of non-strict functional languages is the absence of *closures*. In the traditional implementation of call-by-need, the field arg_i would contain a closure consisting of: (i) a pointer to the code that will compute the i-th parameter, and (ii) an environment, providing the values of the captured variables that this code needs to use. On the other hand, in our implementation, arg_i is just a code pointer. The environment has been eliminated, as the intensional transformation has encoded it in the context (i.e., a pointer to a LAR) that will be passed to arg_i. All variables correspond to top-level, zero-order definitions and it is the context that guides evaluation and produces the correct values of these variables.

The implementation includes certain rather simple optimizations which focus on allocating LARs on the stack whenever this is possible:

– Functions returning ground values (e.g., integers or booleans) or data types with only nullary constructors allocate their LARs on the stack and deallocate them on return.
– Functions that may return data types built by non-nullary constructors allocate their LARs on the heap.

Using this scheme, programs that do not make extensive use of user-defined data types can benefit from stack allocation. Further optimizations are possible, such as tail call elimination, but have not yet been implemented. Usage analysis can also be handy for further optimizations. If, for example, it is known that the value of some arg_i is only used once, then it need not be stored in val_i.

Stack-allocated LARs are discarded immediately when the active function call terminates. On the other hand, a garbage collector is required to discard heap-allocated LARs. We have currently implemented a simple semi-space copying garbage collector but we intend to investigate this further and expect that much better performance can be achieved with a garbage collector more suitable for the nature and usage of LARs; this is one of the primary goals for our future research. The root set for garbage collection is calculated by traversing stack-allocated LARs and the active context.

5 Performance Evaluation

In order to evaluate the performance of our implementation, we benchmarked it against four other well-known Haskell compilers:[1]

– The Glasgow Haskell Compiler (GHC): the definitive compiler for Haskell.
– The Utrecht Haskell Compiler (UHC): implemented using attribute grammars and supporting most features of Haskell 98 and Haskell 2010.
– The NHC98: a small and portable compiler for Haskell 98.
– The JHC: an experimental and fast compiler for Haskell, implemented in order to test various optimizations for the language.

The comparison is based on a set of 13 benchmark programs, most of which are standard benchmarks for lazy functional languages, e.g. coming from the NoFib benchmark suite [11]. Some of the programs perform purely numerical computations (such as the programs `ack`, `fib`, `primes` and `queens-num`), pure list processing (such as `naive-reverse` and `fast-reverse`), numerical computations combined with list-processing and/or higher-order functions (such as `church`,

[1] The code of our implementation and the benchmark programs that we used are available from `http://www.softlab.ntua.gr/~gfour/dftoic/`

Program	`GIC`	`GIC-llvm`	`GHC7`	`GHC6`	`NHC`	`UHC`	`JHC`
ack	2.47	1.25	0.62	0.48	6.18	40.03	0.05
church	3.55	2.09	0.61	0.55	11.58	68.37	0.17
collatz	0.69	0.41	1.07	2.66	84.28	46.90	0.16
digits_of_e1	2.30	2.09	0.77	1.74	60.71	75.29	–[1]
fast-reverse	3.03	1.95	1.74	1.82	1.35	9.41	–[2]
fib	1.35	1.12	0.50	0.51	10.43	55.55	0.17
naive-reverse	3.02	2.87	0.49	0.42	0.79	3.56	0.75
ntak	8.62	5.87	2.91	3.65	154.74	91.95	7.18
primes	2.55	1.58	2.19	2.30	172.45	173.81	0.73
queens-num	0.33	0.23	0.31	0.33	21.16	12.43	0.14
queens	3.92	3.24	0.44	0.48	27.17	123.98	0.82
quick-sort	3.18	2.77	1.92	1.90	1.51	5.42	8.58
tree-sort	2.19	1.97	0.39	0.33	0.91	6.58	0.72
GMR[3]	1.38	1.00	0.51	0.57	7.28	18.49	0.33

[1] `jhc` compilation error, [2] `jhc` runtime error.
[3] Geometric mean of the ratios, compared to `GIC-llvm`.

Fig. 5. Runtime comparison for 13 benchmarks. Execution times are in seconds.

`ntak`, `collatz`, `digits_of_e1`, `quick-sort`), and other user-defined data types (such as `queens` and `tree-sort`).

The benchmarks were performed on a machine with four quad-core Intel Xeon E7340 2.40GHz processors and 16 GB memory, running Debian 6.0.5. The versions of the compilers tested were GHC 7.4.1 and GHC 6.12.1, UHC/EHC 1.1.4, NHC98 1.22, and JHC 0.8.0. Our own compiler is shown in the benchmarks table as `GIC` (the Generalized Intensional Compiler). All benchmarks were executed five times and the median (elapsed) execution time was recorded. For all compilers the effects of garbage collection were minimized by setting a large size for the heap — in practice all programs either did no garbage collection at all or only a few. Finally, we disabled strictness analysis from all compilers that supported such an option, so as to focus on the performance of genuine lazy implementations. As this results in a significant slowdown for compilers like GHC, we will have to repeat the experiment when a competitive strictness analysis has been implemented for our compiler.

The performance results are depicted in Figure 5. In this table, `GIC-llvm` is the generalized intensional compiler whose C output is compiled using `llvm-gcc`, the front-end of `gcc` to the LLVM compiler. We used GCC 4.4.5 and LLVM 2.6. The benchmarks appear to suggest the following conclusions:

– Compiling the target C code of the generalized intensional compiler with `llvm-gcc` is quite more efficient than with standard `gcc`. Very similar results were also obtained using `clang`. In the following, when we refer to the intensional compiler, we mean `GIC-llvm`.
– The intensional implementation is on the average 2-3 times slower than the fully optimized implementations `GHC6` and `GHC7`. Notably, for `collatz`,

`primes`, and `queens-num`, the intensional system performs better than `GHC6` and `GHC7`. Since the intensional compiler does not currently support any sophisticated optimizations, we believe that there is room for much improvement in our implementation.

– In certain programs (e.g., `ack` and `church`) `GHC6` performs better that `GHC7`. This has been reported (ticket #5888 in the GHC bug tracking system); it is related to a GHC optimization for unboxing integer values which seems to have deteriorated in GHC 7. It is expected to be fixed in release 7.6.1.

In general, we feel that the performance results are quite promising for the intensional approach, especially if we take into consideration that it is a far less mature compiler and that its implementation mainly aimed at simplicity and not performance, at this point.

6 Related Work

The work described in this paper, has its roots in the area of *dataflow programming*, which flourished more than three decades ago. It is also connected to the area of *intensional* and *multidimensional programming* [2] which was later developed as an extension of dataflow programming. The proposed technique has its origins in the key ideas that have been developed in order to implement dataflow and intensional languages.

Implementation Techniques for Dataflow Languages. In the dataflow model of computation, data are processed while they are flowing through a network of interconnected nodes (or *dataflow network*). A dataflow network is a system of *processing units* (or *nodes*) which are connected with *communication channels* (or *arcs*). Nodes can have multiple input and output arcs. The most advanced form of dataflow is the so-called *tagged token dataflow* in which the data-items are labeled with *tags* (or *contexts*). A node can fire if it receives in its input arcs data-items that have the same tags. The tagged-token approach obviates the need of data-items to arrive in a strictly pipelined way.

The majority of languages that were used to program dataflow computers were functional in flavor. Therefore, there existed an obvious need to compile recursive functions in a way compatible with the tagged-token model. Many such implementations were developed (e.g., see [9,1]). The key idea of such implementations was to use tags to distinguish data items that belong to different function invocations. This tag-based implementation of recursive functions was known in the dataflow circles as *coloring*. Under the coloring scheme, higher-order functions were implemented by introducing special *apply* nodes in the dataflow graph that used a closure representation for function dispatch [18,12].

The similarity of coloring with the approach proposed in this paper should be apparent by now. Tags correspond to the contexts in our technique. In particular, a context in our technique is used in order to uniquely identify a particular function call in the recursion tree of a program. One can say that the proposed approach transfers the key ideas of dataflow implementations to mainstream lazy functional languages. The novel aspects of our approach are the extension of the coloring technique to a language with user-defined data-types and its efficient implementation on stock hardware.

Intensional Languages and their Implementation. The development of dataflow languages was continued during the nineties with the invention of an extension of dataflow programming, namely *intensional programming* [2]. The first intensional/dataflow language was Lucid [19] whose implementation was based on the original intensional transformation which was formalized through the use of *intensional logic* in A. Yaghi's Ph.D. dissertation [20]. The correctness of the intensional transformation was established in [16]. The novel aspect of the current approach with respect to the original intensional transformation is the support of user-defined data-types and pattern matching.

A recent extension of Lucid is the language TransLucid [13]. The problem of implementing higher-order functions in the context of TransLucid has been considered and the solution that has been proposed is through an explicit representation for closures using extra dimensions (which amount to multiple contexts). To our knowledge, the technique for implementing TransLucid has not been applied to more mainstream functional languages.

Finally, we should note that (to our knowledge) all implementations of intensional languages rely on a runtime structure known as the *warehouse*. The warehouse is a hash-table in which intermediate results are stored in order to be reused when demanded again. Despite the fact that our technique shares the same underlying demand-driven execution model with the intensional languages (since they all rely on the original intensional transformation), our runtime structures and implementation decisions are completely different.

Implementations of Functional Languages. In general, the intensional approach to implementing functional languages appears to differ in philosophy with respect to the graph-reduction-based implementations. The work that appears to be closest to our approach is Boquist's GRIN compiler [3], which is also based on a defunctionalized representation. While GRIN uses a variety of "tags" to characterize different constructs of a lazy language (constructors, function applications, and partial applications), we use a uniform representation for these three types of constructs. GRIN was based on a strict first-order language, in contrast to our source language, FOFL, which is non-strict. Moreover, GRIN directly compiled its language for graph reduction using custom optimizations such as a unique interprocedural register allocation algorithm; we transform it to a zero-order intensional language and compile the intensional representation into C code, using a runtime that is based on lazy activation records.

The generalized intensional transformation has some conceptual similarities with environment-based abstract machines, like the work of Friedman and Wise [7], Henderson and Morris [8], and Krivine [10], or the environment-based STG machines of De La Encina and Peña [5]. One important distinction of the intensional approach with respect to the above, is that our technique is based on a first-order source language. However, one could say that the contexts of our technique play in some sense the role of the environment, since they guide the execution mechanism to perform the correct substitution in the body of a function. We feel that a further investigation of the connections between the two approaches is quite worthwhile.

7 Conclusions and Future Work

We have introduced the *generalized intensional transformation*, an extension of the original intensional transformation that can be used to implement lazy functional languages with user-defined data types. We have demonstrated the usefulness of the proposed technique by implementing such a compiler for a subset of Haskell and by comparing its performance with existing Haskell implementations. There are certain aspects of the technique that appear to require a more extensive investigation:

– Our implementation currently compiles only a fragment of Haskell. It is our intention to extend the implementation to cover the full language. One possibility would be to make our implementation a back-end to GHC, since the GHC core language is (roughly speaking) a higher-order version of our FOFL language. This would allow us to take advantage of all the optimizations and language extensions of GHC.
– Our technique is heavily based on defunctionalization. It is a well-known fact that defunctionalization is a whole-program transformation and therefore one cannot do separate compilation. This is one aspect of our approach which we intend to further investigate. The discussion given in the concluding section of [14] might be a good start on lifting this shortcoming of defunctionalization.
– At present, the compiler only supports a minimal set of optimizations and the runtime system was implemented having simplicity as the driving criterion rather than efficiency. We are currently investigating optimizations at the intensional level and we plan to fine-tune the runtime in order to achieve a better performance. We also intend to investigate the possibility of using LLVM (instead of C) as the compiler's target language.
– We have implemented a simple-minded garbage collection scheme for LARs, which is currently non-portable and not mature enough to be discussed in this paper. We expect the implementation of an efficient garbage collector to be one of the major efforts of our future research, in conjunction with a possible re-implementation of the runtime system.

We feel that the simplicity of the technique and the promising performance results suggest that the intensional approach is worth further consideration as an alternative technique for implementing lazy functional languages.

References

1. Arvind, R.S.N.: Executing a program on the MIT tagged-token dataflow architecture. IEEE Transactions on Computers 39, 300–318 (1990)

2. Ashcroft, E.A., Faustini, A.A., Jagannathan, R., Wadge, W.W.: Multidimensional Programming. Oxford University Press (1995)
3. Boquist, U., Johnsson, T.: The GRIN project: A highly optimising back end for lazy functional languages. In: Kluge, W. (ed.) IFL 1996. LNCS, vol. 1268, pp. 58–84. Springer, Heidelberg (1997)
4. Charalambidis, A., Grivas, A., Papaspyrou, N.S., Rondogiannis, P.: Efficient intensional implementation for lazy functional languages. Mathematics in Computer Science 2(1), 123–141 (2008)
5. De La Encina, A., Peña, R.: From natural semantics to C: A formal derivation of two STG machines. Journal of Functional Programming 19(1), 47–94 (2009)
6. Fourtounis, G., Papaspyrou, N., Rondogiannis, P.: The intensional transformation for functional languages with user-defined data types. In: Proceedings of the 8th Panhellenic Logic Symposium, pp. 38–42 (2011)
7. Friedman, D.P., Wise, D.S.: CONS should not evaluate its arguments. In: Proceedings of the International Colloquium on Automata, Languages and Programming, pp. 257–284 (1976)
8. Henderson, P., Jr. Morris, J.H.: A lazy evaluator. In: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages, pp. 95–103. ACM, New York (1976)
9. Kirkham, C., Gurd, J., Watson, I.: The Manchester prototype dataflow computer. Communications of the ACM, 34–52 (1985)
10. Krivine, J.L.: Un interpréteur du lambda-calcul, `http://www.pps.univ-paris-diderot.fr/~krivine/articles/interprt.pdf`
11. Partain, W.: The nofib benchmark suite of Haskell programs. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming, pp. 195–202 (1993)
12. Pingali, K.: Lazy evaluation and the logic variable. In: Proceedings of the 2nd International Conference on Supercomputing, pp. 560–572. ACM, New York (1988)
13. Plaice, J., Mancilla, B.: The practical uses of TransLucid. In: Proceedings of the 1st International Workshop on Context-aware Software Technology and Applications, pp. 13–16. ACM, New York (2009)
14. Pottier, F., Gauthier, N.: Polymorphic typed defunctionalization and concretization. Higher-Order and Symbolic Computation 19, 125–162 (2006)
15. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the 25th ACM National Conference, pp. 717–740. ACM (1972)
16. Rondogiannis, P., Wadge, W.W.: First-order functional languages and intensional logic. Journal of Functional Programming 7(1), 73–101 (1997)
17. Rondogiannis, P., Wadge, W.W.: Higher-order functional languages and intensional logic. Journal of Functional Programming 9(5), 527–564 (1999)
18. Traub, K.R.: A compiler for the MIT tagged-token dataflow architecture. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1986)
19. Wadge, W., Aschroft, E.A.: Lucid, the Dataflow Programming Language. Academic Press (1985)
20. Yaghi, A.A.: The Intensional Implementation Technique for Functional Languages. Ph.D. thesis, Department of Computer Science, University of Warwick, Coventry, UK (1984)

`Terminyzer`: An Automatic Non-termination Analyzer for Large Logic Programs[⋆]

Senlin Liang and Michael Kifer

Department of Computer Science
Stony Brook University, USA
{`sliang,kifer`}`@cs.stonybrook.edu`

Abstract. There have been many studies on termination analysis of logic programs but little has been done to analyze their *non*-termination, an even more important task, in our opinion. Non-termination analysis examines program execution history when non-termination is suspected and attempts to inform the programmer about possible ways to fix the problem. This paper attempts to fill in the void. We study the problem of non-termination in tabled logic engines with subgoal abstraction, such as XSB, and propose a suite of algorithms, called non-*Termin*ation Anal*yzer*, `Terminyzer`, for automatic *detection* of non-termination and *explaining* it to the user. `Terminyzer` includes several non-termination analysis approaches of different computational complexity. These approaches are all based on analyzing forest logging traces and supply sequences of tabled calls that are likely causes of non-terminating cycles. It also provides the sequences of functors that are applied repeatedly to generate infinitely many answers and thus helps programmers debug large programs by focusing on much smaller subsets of rules.

`Terminyzer` is included in both XSB and $\mathcal{F}$LORA-*2*, and all examples used in this paper are available online[1].

Keywords: non-termination analysis, termination analysis, logic programming, forest logging, tabling, subgoal abstraction.

1 Introduction

The development of high-level logic languages such as $\mathcal{F}$LORA-*2* and SILK aims at making logic-based knowledge representation accessible to knowledge engineers who are not programmers. This type of users cannot be expected to debug the procedural aspects of the rule bases that they create and thus they require special support. In the course of the SILK project we discovered that non-termination due to the use of HiLog [2] and function symbols is one of the most vexing problems such users are facing, which motivated the present work.

[⋆] This work is part of the SILK (Semantic Inference on Large Knowledge) project sponsored by Vulcan, Inc. The authors were also partially supported by the NSF grant 0964196. The basic idea behind `Terminyzer` was also mentioned in [6].

[1] `www.cs.sunysb.edu/~sliang`

There are three main scenarios where programs may not terminate. First, recursion can cause non-termination under the usual Prolog evaluation strategy. For instance, the following simple program will not terminate in Prolog:

```
p(X) :- p(X).
?- p(a).
```

This kind of problems have been successfully addressed by adding SLG resolution (also known as tabling) to Prolog, and a number of systems support it to various degrees (XSB [17], Yap [3], B-Prolog [21], Ciao [5]).

The second scenario where programs might not terminate, even under SLG resolution, occurs when increasingly deep nested calls are generated during the evaluation. Consider the following simple program:

```
:- table p/1.
p(X) :- p(f(X)).
?- p(a).
```

In this case, the following calls will be successively generated: `p(a)`, `p(f(a))`, `p(f(f(a)))`, and so on. Since neither call subsumes the other, tabling will not be able to evaluate the query and terminate. However, the technique known as *subgoal abstraction* can take care of this problem. The essence of the technique is to modify the calls by replacing ("abstracting") subterms with new variables once certain term depth limit has been reached. For instance, in our example we could abstract calls once the depth limit of 4 has been reached. As a result, `p(f(f(f(f(a)))))` and all the subsequent calls would be abstracted to `p(f(f(f(X))))`.

For instance, in XSB (which to our knowledge is the only system that supports both tabling and subgoal abstraction), the above program will terminate. Generally, tabling with subgoal abstraction will evaluate queries that have finite number of answers. Thus, the only remaining scenario is when both tabling and subgoal abstraction are used, but query evaluation does not stop because the number of answers to the query or its subqueries is infinite. An example one program that exhibits this behavior is

```
:- table p/1.
p(a).
p(f(X)) :- p(X).
?- p(X).
```

where the query has the answers `p(a)`, `p(f(a))`, `p(f(f(a)))` and so on.

In general, such queries cannot be evaluated completely, but if the program is what the user intended, the user could ask the system to stop after getting the first few answers. The problem arises when this was not the intended result. For small programs with only a few rules, expert programmers might be able to find the causes of the problem. However, for large knowledge bases with hundreds or thousands of rules this becomes a difficult task even for a seasoned logic programmer. For a knowledge engineer who is not a programmer, debugging non-termination is out of the question.

Note that neither program termination (the halting problem) nor the problem of whether the number of answers is finite is decidable [14,15], so no algorithm can provide guarantees of termination or to prove non-termination in general. Sufficient conditions for termination of logic programs have been proposed in the literature [14,18,8,1,10,11,13], but most deal with Prolog or Prolog-like evaluation strategies. Neither tabling nor subgoal abstraction were taken into account, so these works have very limited use for advanced logic engines like XSB and its derivatives, $\mathcal{F}$LORA-*2* and SILK. This paper therefore takes a different track on the problem: developing techniques that can help users analyze the causes of non-termination. We introduce a suite of algorithms, called *Non-termination Analyzer*, or `Terminyzer`, which are based on the analysis of logs produced by table operations for calls to tabled predicates. The algorithms report the potential causes of non-termination with increasing level of fidelity and precision. Of course, the higher-fidelity algorithms have higher complexity.

The paper is organized as follows. Section 2 provides the basics of tabling and forest logging in XSB. Section 3 introduces three algorithms underlying `Terminyzer` and discusses their complexity. Section 4 describes the experimental studies, and Section 5 discusses related work and concludes the paper.

2 Tabling and Forest Logging in XSB

The limitations of the standard SLD resolution-based evaluation strategy used in Prolog are well-known: it is woefully incomplete and can go into an infinite loop even for simple Datalog rule sets. To address this limitation, *SLG resolution* (also known as "tabling") was developed and implemented [17]. In tabled evaluation, calls to certain predicates, which are declared as *tabled*, are cached in a *table* $\mathcal{T}$ to be used by subsequent calls. $\mathcal{T}$ can be viewed as a set of pairs of the form $(call, answers)$ where $answers$ are proven instances of $call$.

When a tabled call, $call$, is issued, SLG examines whether there is a pair $(call', answers') \in \mathcal{T}$ such that $call$ is *similar* (to be explained shortly) to $call'$. If so, then $answers'$ are used to satisfy $call$ and no clause resolution is performed for $call$. In this case, $call$ is referred to as the *consumer* of $call'$ while $call'$ is the *producer* of $call$. Otherwise, a new table entry of the form $(call, answers)$ is added to $\mathcal{T}$, where initially $answers = \emptyset$. Then $call$ is resolved against program clauses as usual in Prolog. All newly derived answers for $call$ are added to $answers$, $call$ becomes a producer of these answers, and all subsequent calls that are similar to $call$ become consumers of $call$'s answers.

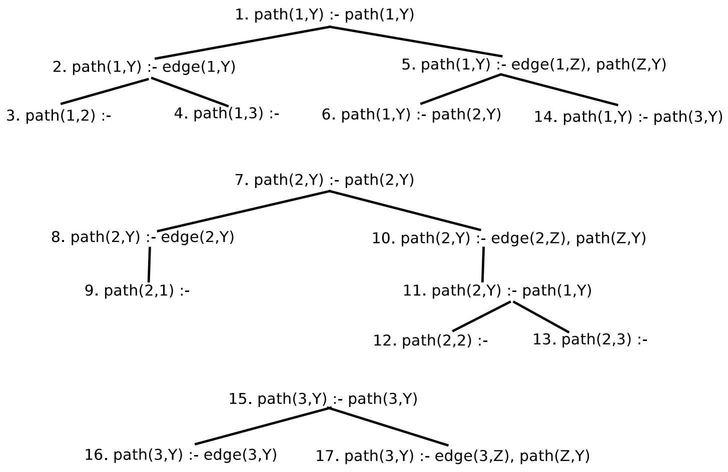
There are two main ways to define call-similarity mentioned above. Depending on which notion is chosen, the tabling strategy is called *variant* or *subsumptive*. In variant tabling, $call$ is similar to $call'$ if $call$ is a *variant* of $call'$, i.e., they are identical up to variable renaming. In subsumptive tabling, $call$ is similar to $call'$ if $call$ is *subsumed* by $call'$, i.e., there is a variable substitution σ such that $\sigma(call') = call$. Note that in this case the notion of similarity is asymmetric. Since only unique answers are added to the table and returned to consumers, tabled evaluation terminates *if* there is only a finite number of tabled calls and each tabled call has finitely many answers. For instance, this is the case in Datalog, i.e., when function symbols are not present. It has been proven that tabled evaluation terminates for any program with the *bounded term depth property*, i.e., all terms that are ever generated in the course of SLG resolution, including all calls and answers, have an upper bound on their depth [17].

The workings of SLG resolution can be captured by an *SLG forest*, which has an *SLG tree* for every new (dissimilar) call to a tabled predicate. The SLG tree for *call* has root of the form *call* :- *call*, and each non-root node is of the form $\theta(call)$:- $\theta(subgoals)$, where $\theta(subgoals)$ are the remaining calls needed to prove *call* and θ is the substitution obtained from resolving *call* against the knowledge base. If $\theta(subgoals)$ is an empty clause, $\theta(call)$ is an answer to *call.* Each edge in the tree corresponds to a derivation step of clause resolution or an answer lookup by consumers.

Example 1. The SLG forest for the following program is shown in Figure 1, where each node is labeled with an ordinal denoting the creation order of the node during evaluation.

```
:- table path/2.
edge(1,2).  edge(1,3).  edge(2,1).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
?- path(1,Y).
```

This is a simplified version of an example in [16]. □



1. path(1,Y) :- path(1,Y)
2. path(1,Y) :- edge(1,Y)
5. path(1,Y) :- edge(1,Z), path(Z,Y)
3. path(1,2) :-
4. path(1,3) :-
6. path(1,Y) :- path(2,Y)
14. path(1,Y) :- path(3,Y)
7. path(2,Y) :- path(2,Y)
8. path(2,Y) :- edge(2,Y)
10. path(2,Y) :- edge(2,Z), path(Z,Y)
9. path(2,1) :-
11. path(2,Y) :- path(1,Y)
12. path(2,2) :-
13. path(2,3) :-
15. path(3,Y) :- path(3,Y)
16. path(3,Y) :- edge(3,Y)
17. path(3,Y) :- edge(3,Z), path(Z,Y)

Fig. 1. The SLG Forest for Example 1

Compared to Prolog systems, logic engines that support tabling are much more involved. They suspend and resume computation paths, delay negated calls that are involved in loops through negation, simplify these calls once their truth values become known, and manage the table accordingly. For debugging and performance optimization, programmers sometimes need to inspect table operations during evaluation. To this end, XSB provides forest logging (`logforest`), which makes the table events available to the programmer. These events include:

- *Call to a tabled predicate.* When a subgoal call, *child*, is made during the evaluation of *parent*, a Prolog fact of the form $tc(child, parent, status, counter)$ is logged. Here *counter* is an ordinal representing the unique id and sequence order of the above event and *status*, the current status of *child*, is
  - *new* if *child* is a newly issued call;
  - *cmp* if the evaluation of *child* has been completed; and
  - *incmp* if *child* is not a new call, but is yet to be completely evaluated.

  If *child* is the first tabled call in an evaluation, *parent* is the node *root*.
- *Derivation of a new answer.* When a new answer *ans* is derived for *call* and added to the table, the fact $na(ans, call, counter)$ is added to the log.
- *Return of an answer to a consumer.* If an answer *ans* is returned to a consumer *child* which is issued during evaluation of *parent*, the fact $ar(ans, child, parent, counter)$ is added to the log.
- *Subgoal completion.*
  - When a set $\mathcal{S}$ of mutually recursive subgoals are completely evaluated, `logforest` records $cmp(call, sccnum, counter)$ for each $call \in \mathcal{S}$, where *sccnum* is an ordinal that identifies this set of mutually recursive calls.
  - If a subgoal *call* is *completed early* (i.e., its truth is established without the need to fully evaluate all the dependent subgoals), $cmp(call, ec, counter)$ is logged where *ec* stands for *early completion.*
- *Table abolishes and errors.* These events correspond to user-requests to abolish parts of the table or to errors issued by the evaluation mechanism. Such events are not needed for our purposes and we will omit them in the sequel.

Example 2. For the SLG forest of Example 1, the `logforest` trace is given in the first column of Table 1. The second column in the table is the label of the node in the tree of Figure 1 where a corresponding event happens. The third column is an explanation. An answer for a call is represented as a substitution for the list of variables in the call. For instance, in the second log entry $na([2], path(1, \_v0), 1)$, the answer is represented as [2] and the list of variables in the call $path(1, \_v0)$ are $[\_v0]$. It means that the substitution $\_v0 = 2$ is an answer. □

3 `Terminyzer`: Non-termination Analyzer

Since `logforest` records only table operations during evaluation and is implemented in C, it works much faster and produces much smaller logs compared with traditional tracing facilities [16]. This makes it possible to analyze forest logging traces and thus help to debug large programs. Recall that with SLG and subgoal abstraction, the only scenario where query evaluation may not terminate is when it or its subqueries have an infinite number of answers.

Table 1. Forest Log for the Evaluation of Example 1

Log	Label	Explanation
`tc(path( 1,_v0),root,new,0)`	1	initial call
`na([ 2],path( 1,_v0),1)`	3	new answer
`na([ 3],path( 1,_v0),2)`	4	new answer
`tc(path( 2,_v0),path( 1,_v0),new,3)`	7	new call made by node 6
`na([ 1],path( 2,_v0),4)`	9	new answer
`tc(path( 1,_v0),path( 2,_v0),incmp,5)`	11	repeated incomplete call
`ar([ 2],path( 1,_v0),path( 2,_v0),6)`	12	table look up, answer to consumer
`na([ 2],path( 2,_v0),7)`	12	new answer
`ar([ 3],path( 1,_v0),path( 2,_v0),8)`	13	table look up, answer to consumer
`na([ 3],path( 2,_v0),9)`	13	new answer
`tc(path( 3,_v0),path( 1,_v0),new,10)`	15	new call made by node 14
`cmp(path( 3,_v0),3,11)`	15	evaluation completed
`ar([ 1],path( 2,_v0),path( 1,_v0),12)`	9	return to consumer
`na([ 1],path( 1,_v0),13)`	6	new answer
`ar([ 2],path( 2,_v0),path( 1,_v0),14)`	12	return to consumer
`ar([ 3],path( 2,_v0),path( 1,_v0),15)`	13	return to consumer
`ar([ 1],path( 1,_v0),path( 2,_v0),16)`	6	return to consumer
`cmp(path( 1,_v0),1,17)`	1	evaluation completed
`cmp(path( 2,_v0),1,18)`	7	evaluation completed

In this section, we describe three algorithms from the non-termination analysis suite `Terminyzer`. These algorithms target different aspects of non-termination and are useful separately and in combination—they are incomparable in terms of their usefulness. However, of the three, the first method, call sequence analysis, is both sound and complete, and is computationally less expensive. The other two algorithms are only *complete*, i.e., they enumerate all causes of non-termination, but some of the causes that they flag may turn out to be false-positives.

The algorithms are based on stopping the execution after a time limit set by the user or after the evaluation starts producing answers that exceed certain size limits, and then analyzing the logs. Our examples assume that the system stops after generating query answers of depth greater than 5. We emphasize that due to the undecidability results mentioned in the introduction, one cannot algorithmically prove non-termination in all cases unless infinite logs are available. Pragmatically, this means that, in working with `Terminyzer`, one must assume that the available logs are "long enough."

3.1 Call Sequence Analysis

Call sequence analysis finds sequences of calls to tabled predicates and each such sequence is a potential cause for non-termination. As discussed in Section 2, when a call to tabled predicate has been completely evaluated and all its answers have been recorded in the table, `logforest` dumps a corresponding log entry of the form *cmp*(*call*, *sccnum*, *counter*). We say that such calls are *finished.* Otherwise, the call is *unfinished*. We write $unfinished\_call(child, parent, counter)$ to indicate that the call *child* has been issued in the SLG tree for *parent* by an event with id *counter* but the call is still unfinished. Since *parent* is waiting for the answers from the table created for *child*, *parent* is a child of another unfinished call. The initial call, *root*, has no parent.

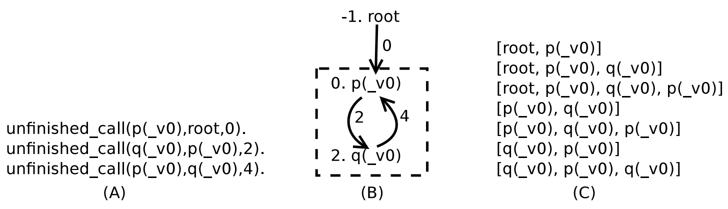
The *unfinished-call child-parent graph* (CPG) for a forest logging trace is a directed graph $G_{unfinished} = (\mathcal{N}, \mathcal{E})$ where the set of nodes is $\mathcal{N} = \{child \mid unfinished\_call(child, parent, counter)\} \cup \{root\}$. Each node $call \in \mathcal{N}$ is *labeled* with the *counter* value of the event that originally issued *call*; this label is written as $call.label$. The label of initial call *root* is -1. A directed edge $(call_1, call_2) \in \mathcal{E}$ exists in $G_{unfinished}$ if and only if $call_1$ is an unfinished call parent of $call_2$, i.e., $unfinished\_call(call_2, call_1, counter)$ is true. Also, the edge that corresponds to $unfinished\_call(call_2, call_1, counter)$ is *labeled* by $counter$. The labels of nodes and edges preserve the temporal order of their creation in forest logging trace.

An *unfinished-call path* is a path with *no repeated edges* in $G_{unfinished}$; it is called an *unfinished-call loop* if it is a cycle. An unfinished-call path of the form $[call, call]$ means that there is an edge $(call, call) \in \mathcal{E}$ and it is also an unfinished-call loop. Loops that represent the same cycles in CPG are considered to be the same and we keep only one representative for each such set of loops. For instance, $[a, b, c, a]$ and $[b, c, a, b]$ are the same loop while $[a, b, c, a]$ and $[a, c, b, a]$ are not.

Example 3. The evaluation of query `?- p(X)` against the following program

```
:- table p/1, q/1.
p(a).  q(b).
p(f(X)) :- q(X).  q(g(X)) :- p(X).
```

will produce logs containing unfinished calls shown in Figure 2(A) where the calls are sorted according to *counter* values. The corresponding unfinished-call CPG is depicted in Figure 2(B) with each node and edge labeled. The unfinished-call CPG has seven unfinished-call paths shown in Figure 2(C); $[p(\_v0), q(\_v0), p(\_v0)]$ and $[q(\_v0), p(\_v0), q(\_v0)]$ are also unfinished-call loops (they are marked with a square in Figure 2(B)). Since they are the same loop, we keep only one of them, $[p(\_v0), q(\_v0), p(\_v0)]$. □



```
unfinished_call(p(_v0),root,0).
unfinished_call(q(_v0),p(_v0),2).
unfinished_call(p(_v0),q(_v0),4).
```

(A)

```
[root, p(_v0)]
[root, p(_v0), q(_v0)]
[root, p(_v0), q(_v0), p(_v0)]
[p(_v0), q(_v0)]
[p(_v0), q(_v0), p(_v0)]
[q(_v0), p(_v0)]
[q(_v0), p(_v0), q(_v0)]
```

(C)

Fig. 2. The Unfinished Call Loop of Example 3

Theorem 1 (Completeness). *Consider a query and a program all of whose predicates are tabled and assume that the system supports subgoal abstraction. If the evaluation does not terminate, then there exists at least one unfinished-call loop in the unfinished-call child-parent graph for its complete infinite forest logging trace.* □

The proof of Theorem 1 is omitted due to space limitation and can be found in [7]. Clearly, one cannot obtain the complete infinite trace for a non-terminating evaluation. In practice, one would let the program execute long enough until it starts producing answers exceeding some size limits and then analyze the available portion of the log.

Algorithm 1 constructs the unfinished-call CPG $G_{unfinished} = (\mathcal{N}, \mathcal{E})$ from the set of unfinished calls of a forest logging trace. For each $unfinished\_call$ $(child, parent, counter)$, the node *child* is added, if it does not already exist, plus the edge $(parent, child)$. All unfinished calls are processed in the order of their creation in forest logging, which is also the order in which these unfinished calls are issued during evaluation. Thus, when $unfinished\_call(child, parent, counter)$ is encountered we know that *parent* must have been added to the graph as an unfinished child call of some other parent, i.e., $unfinished\_call(parent, p', counter')$ must be true for some p' and $counter' < counter$. We have two cases:

1. $child \in \mathcal{N}$. The evaluation calls a previously issued subgoal.
2. $child \notin \mathcal{N}$. A new subgoal is called and a new node is added to the graph.

In the first case, an unfinished-call loop exists, so the current evaluation path of *parent* is suspended and alternative derivations will be explored. This implies an important property of unfinished-call CPGs: an unfinished-call loop is created out of an (acyclic) path always by adding a final edge of the form $(call_1, call_2)$, where $call_1.label \geq call_2.label$. We call such an edge a *critical loop edge*.

```
1 Let $UnfinishedCalls$ be the set of all unfinished calls;
2 $\mathcal{E} = \emptyset$;  $\mathcal{N} = \{root\}$;  $root.label = -1$;
3 while $UnfinishedCalls \neq \emptyset$ do
4     Remove $unfinished\_call(child, parent, counter)$, where $counter$ is the
      smallest among $UnfinishedCalls$;
5     if $child \notin \mathcal{N}$ then  $\{\mathcal{N} = \mathcal{N} \cup \{child\};\ child.label = counter\}$;
6     $\mathcal{E} = \mathcal{E} \cup \{(parent, child)\}$;  $(parent, child).label = counter$;
7 end
8 return $G_{unfinished} = (\mathcal{N}, \mathcal{E})$
```

Algorithm 1. Unfinished-Call CPG Construction

If critical loop edges are taken out, any unfinished-call CPG becomes a connected directed acyclic graph in which every edge goes from a node with a smaller label to a node with a larger label. A path, *path*, connecting the root node *root* to a call, *call*, exists if and only if the evaluation process issues a sequence of calls, as specified by the edges in *path*. There is at least one such path for every $call \in \mathcal{N}$.

After computing all unfinished-call paths from *root* to other nodes, all distinct unfinished-call loops can be computed by checking whether there are repeated vertices in each path. Consider an unfinished-call path $P = [root, call_1, \ldots, call_n]$. If $call_n = call_i$, $1 \leq i \leq n$, then the part of P from $call_i$ to $call_n$ $[call_i, \ldots, call_n]$ is an unfinished-call loop. In this case, P is said to contain an unfinished-call loop and we also know that P is a sequence of calls that may have caused non-termination.

Example 4. The unfinished-call CPG in Figure 2(B) of Example 3 is constructed from the calls in Figure 2(A) using Algorithm 1. The third edge, $(q(\_v0), p(\_v0))$, is the only critical loop edge. The unfinished-call paths starting from *root* are $P_1 = [root, p(\_v0)]$, $P_2 = [root, p(\_v0), q(\_v0)])$, and $P_3 = [root, p(\_v0), q(\_v0), p(\_v0)])$. Since P_3 contains a repeated vertex $p(\_v0)$, $[p(\_v0), q(\_v0), p(\_v0)]$ is an unfinished-call loop and and thus this sequence of calls in P_3 may be responsible for non-termination. □

3.2 Soundness of Call Sequence Analysis

One problem with the previous algorithm for call sequence analysis is that finding all unfinished-call paths in an unfinished-call CPG is in NP. Even if it were polynomial, we could practically present only 2 or 3 paths to the user without risking to overwhelm him and making the technique useless. Furthermore, these paths better be useful for identifying sources of non-termination: Theorem 1 guarantees only the completeness of call-sequence analysis, which means it can give us false-positives.

We say that an unfinished-call path or loop is *culprit* if it is a cause for non-termination. This section presents a linear algorithm to find one culprit unfinished-call path in an unfinished-call CPG $G_{unfinished} = (\mathcal{N}, \mathcal{E})$.

Let $call_{max} \in \mathcal{N}$ be the node with the maximal label. We will show that $call_{max}$ is contained in a culprit unfinished-call loop. If $call_{max}$ is not part of any such loop, then all the culprit unfinished-call loops in $G_{unfinished}$ contain only calls that are issued before $call_{max}$. We also know that there exists at least one culprit unfinished-call loop when non-termination happens. This means that non-termination happens before $call_{max}$ is issued and in fact $call_{max}$ should never have been issued at all. This contradicts the fact that $call_{max} \in \mathcal{N}$ and, thus, $call_{max}$ must be contained in a culprit unfinished-call loop.

Algorithm 2 finds a culprit unfinished-call path, *path*, from *root* to $call_{max}$. It starts with $child = call_{max}$ and follows reverse directions of edges until reaching *root*. During each while-loop iteration from line 2 to 6, *child* is the current call under consideration and *path* is an unfinished-call path from *child* to $call_{max}$ which is also part of the culprit unfinished-call path to be computed. The parent of *child*, $parent_{max}$, chosen in line 3 satisfies two conditions:

1. $parent_{max}.label < child.label$ because $parent_{max}$ must have been first issued prior to *child*; and

2. $(parent_{max}, child).label = \max\{label \mid (parent, child).label\}$, where $(parent, child).label$ is the label assigned to the edge $(parent, child)$. This means that among all edges going into *child*, $(parent_{max}, child)$ has the maximal label.

These two conditions tell us that the call *child* is issued from the SLG tree for $parent_{max}$, and this is the *last* time that *child* was called during evaluation of the previous calls to it. We know that $parent_{max}$ must be in a culprit unfinished-call path of the form $[root, ..., parent_{max}, child, ..., call_{max}]$, since otherwise non-termination must have happened *before child* was called from the SLG tree for $parent_{max}$. In this case, the call to *child* should have never been made at all—contrary to the assumption that $(parent_{max}, child) \in \mathcal{E}$.

```
1 $path = [call_{max}]$;  $child = call_{max}$;
2 while $child \neq root$ do
3     Let $parent_{max}$ be a parent of $child$ such that $parent_{max}.label < child.label$
      and $(parent_{max}, child).label = \max\{label \mid (parent, child).label\}$;
4     $path = [parent_{max} | path]$;
5     $child = parent_{max}$;
6 end
7 return $path$
```

Algorithm 2. Culprit Unfinished-Call Path Computation

Consider an edge $(parent, child) \in \mathcal{E}$, it will only be examined once in line 3 when *child* is under consideration. Therefore, Algorithm 2 is linear in the graph size. We have the following theorem:

Theorem 2 (Soundness). *If Algorithm 2 finds a culprit path then the computation is non-terminating.*

3.3 Answer Flow Analysis

Call sequence analysis begins with the initial call *root* and follows the sequences of unfinished calls produced by query evaluation. However, it was not designed to identify the patterns among these unfinished calls. Answer flow analysis looks for the log entries that specify the answers being returned to parents (the *ar*-facts) and produces patterns of calls for which answers are still being produced. It also tracks how information contained in these answers flows among these calls.

When non-termination happens because the number of answers is infinite, each new answer, *answer*, to an unfinished call, *call*, is returned to the parents of *call* and these parents use *answer* to derive their own answers. The newly derived answers for the parents of *call* are returned to the parents of the parents, and this gives rise to an endless process in which calls continue to receive, derive and return answers. Answer flow analysis detects such child-parent relationships among calls by analyzing the logs for answers returned to parents at the end of the `logforest` trace.

Given a logforest trace, we can find the maximal *counter* of subgoal completions as $\max\{counter \mid cmp(call, sccnum, counter)\}$. It makes sense to consider only the trace entries whose *counter* value is larger than the maximal subgoal completion counter, since only these entries can possibly be involved in the endless process of returning answers to parents. We consider only these log entries in the rest of this section.

Answer flow child-parent sequence is the sequence of child-parent pairs found in all the log entries for answers returned to parents (the *ar*-facts). The pairs in the sequence are sorted by their creation order *counter*. A child might continue returning multiple answers to a certain parent before the parent starts deriving its own answers. In this case, only one child-parent pair is recorded for all such answers, since all these pairs are identical.

An answer-flow child-parent sequence, *cps*, contains a *child-parent pattern* of length n, denoted $cpp = [pair_1, \ldots, pair_n]$, if the sequence *cpp* repeats at least twice at the end of *cps*, i.e., $cps = [\ldots, pair_1, \ldots, pair_n, pair_1, \ldots, pair_n]$. For instance, $[(c_2, p_2), (c_3, p_3)]$ is a child-parent pattern of length two in $[(c_1, p_1), (c_2, p_2), (c_3, p_3), (c_2, p_2), (c_3, p_3)]$. Thus, every child-parent sequence *cps* has a *prefix* that does *not* end with a *cpp* and a *suffix* that consists of two or more repeated *cpp*'s. We will call this suffix the *cpp*-suffix of that *cps*. The *optimal child-parent pattern* in a child-parent sequence *cps* is the *shortest* child-parent pattern, *cpp*, such that its *cpp*-suffix is the longest in *cps* (among all suffixes of child-parent patterns in *cps*). We use $optimal\_cpp(child, parent)$ to denote the fact that $(child, parent)$ is in the optimal child-parent pattern. In case of non-termination, there exists an optimal child-parent pattern, as stated in Theorem 3, below.

Example 5. The forest logging trace for Example 3 has the child-parent sequence $[(p(\_v0), q(\_v0)),\ (q(\_v0), p(\_v0)),\ (p(\_v0), q(\_v0)),\ (q(\_v0), p(\_v0)),\ (p(\_v0), q(\_v0)), (q(\_v0), p(\_v0)),\ (p(\_v0), q(\_v0)),\ (q(\_v0), p(\_v0)),\ (p(\_v0), q(\_v0)), (q(\_v0), p(\_v0))]$. It has two child-parent patterns:
$cpp_1 = [(p(\_v0),\ q(\_v0)),\ (q(\_v0),\ p(\_v0)),\ (p(\_v0),\ q(\_v0)), (q(\_v0), p(\_v0))]$
of length four, twice repeated;
$cpp_2 = [(p(\_v0), q(\_v0)), (q(\_v0), p(\_v0))]$ of length two, repeated five times.

The optimal child-parent pattern is cpp_2, as it covers $2 \times 5 = 10$ entries in *cps* compared to cpp_1, which covers only $4 \times 2 = 8$ entries. □

As in the call sequence analysis, child-parent relationships are modeled as a directed graph. An *answer-flow child-parent graph* for a forest logging trace is a directed graph $G_{answer} = (\mathcal{N}, \mathcal{E})$, defined as follows. Let $cpp_{optimal}$ be the optimal child-parent pattern for the forest logging trace in question. Then $\mathcal{N}$ is the set of children and parent-calls in $cpp_{optimal}$, i.e., $\mathcal{N} = \{call \mid (call, ...) \in cpp_{optimal}$ or $(..., call) \in cpp_{optimal}\}$. Edges in G_{answer} are the child-parent pairs in $cpp_{optimal}$, i.e., $\mathcal{E} = \{(child, parent) \mid (child, parent) \in cpp_{optimal}\}$.

A path in G_{answer} is called an *answer-flow path*; such a path is called an *answer-flow loop* if it is a cycle. Two answer-flow loops that consist of the same nodes and edges are considered to be the same and we will keep only one representative of the loop in such a case. Answer-flow paths depict how information flows among calls.

The answer-flow CPG for a forest logging trace can be constructed from its optimal child-parent pattern $cpp_{optimal}$ similarly to the unfinished-call CPG construction in Algorithm 1. All answer-flow paths and loops can then be computed.

Example 6. The optimal child-parent pattern in Example 5 is $cpp_{optimal} = [(p(\_v0), q(\_v0)), (q(\_v0), p(\_v0))]$. Its answer-flow graph is the subgraph shown inside the rectangle in Figure 2(B). There are four answer-flow paths: $[p(\_v0), q(\_v0)]$, $[p(\_v0), q(\_v0), p(\_v0)]$, $[q(\_v0), p(\_v0)]$, and $[q(\_v0), p(\_v0), q(\_v0)]$; $[p(\_v0), q(\_v0), p(\_v0)]$ and $[q(\_v0), p(\_v0), q(\_v0)]$ represent the same answer-flow loop.

Theorem 3 (Completeness). *Consider a query to a program all of whose predicates are tabled. As before, assume that the inference engine supports subgoal abstraction. If the query evaluation does not terminate, then:*

i. There exists an optimal child-parent pattern in its complete infinite trace,
ii. $G_{answer} = (\mathcal{N}, \mathcal{E})$ contains at least one answer flow loop, and
iii. Every call $\in \mathcal{N}$ appears in at least one answer-flow loop. □

The proof of Theorem 3 can be found in [7]. In the call sequence analysis, an unfinished-call CPG is constructed, all distinct unfinished-call loops are computed, and the suspected unfinished-call loops emanating from *root* are flagged. Similarly, in answer-flow analysis, one builds answer-flow CPG and computes answer-flow loops, which shed light on how answers flow among subgoals when non-termination happens. The following Theorem 4 connects these two analytic approaches. Again, the proof is found in [7].

Theorem 4. *Let $G_{unfinished} = (\mathcal{N}_{unfinished}, \mathcal{E}_{unfinished})$ be the unfinished-call CPG and $G_{answer} = (\mathcal{N}_{answer}, \mathcal{E}_{answer})$ be the answer-flow CPG for a non-terminating forest logging trace. Then $\mathcal{N}_{answer} \subset \mathcal{N}_{unfinished}$, and for every edge $(child, parent) \in \mathcal{E}_{answer}$ there is an edge $(parent, child) \in \mathcal{E}_{unfinished}$. Furthermore, any answer-flow loop is also an unfinished-call loop.* □

3.4 Functor Pattern Analysis

The answer-flow analysis produces an optimal child-parent pattern, whose repetitions cover the tail of a forest logging trace. During each such repetition, new answers are derived and returned to their parents. Thus, these repetitions divide the log entries into *answer segments*, where each segment contains log entries for answers derived during one repetition. *Functor pattern analysis* attempts to distill sequences of functor application that are repeatedly applied in each segment and thus are responsible for the ability of those repeated segments to produce more and more answers.

Let $as_1, \ldots, as_n$ be answer segments of a forest logging trace, and $answers(as_i)$ be the set of answers contained in as_i. We say that an answer $ans_i \in answers(as_i)$ is *constructed out of* an answer $ans_j \in answers(as_j)$, $j < i$, if ans_j is a *subterm* of ans_i, and the sequence of functors that are applied to derive ans_i from ans_j

are called the *increment* from ans_j to ans_i, denoted by $inc(ans_j, ans_i)$. Checking subterm relationship between two terms is an expensive operation which depends on how terms are stored and indexed. So, in general, functor pattern analysis is impractical. However, for *safe*[2] knowledge bases, we can approximate *subterm* matching with a much more efficient operation of *substring* matching.

If an answer, ans_i, is constructed out of another answer ans_j, we know that $str(ans_j)$ is a substring of $str(ans_i)$ where $str(...)$ is the string representation of a term. That is, $str(ans_i)$ can be represented as $inc\_str \cdot str(ans_j) \cdot tail\_str$, where "·" denotes the string concatenation operation. The sequence of functors contained in inc_str is an approximation of $inc(ans_j, ans_i)$. We call the approximation contained in the *shortest* such inc_str the *optimal approximation* of $inc(ans_j, ans_i)$, written as $inc^*(ans_j, ans_i)$. For instance, the two approximations of $inc(f_4(a),\ f_1(f_2(f_4(a)), f_3(f_4(a))))$ are $[f_1, f_2]$ and $[f_1, f_2, f_4, f_3]$; $[f_1, f_2]$ is the optimal approximation.

Assume that $all\_answers(i)$ is the sequence of answers contained in all answer segments as_j such that $j \leq i$, and $all\_answers(i)$ is sorted by creation orders of its answers in forest logging trace. To compute the increment for an answer $ans_i \in answers(as_i)$, functor pattern analysis tries to find the *last* answer $ans \in all\_answers(i\text{-}1)$, such that ans_i is constructed out of ans. If ans is found, we say that the optimal increment approximation $inc^*(ans, ans_i)$ is the *best-fit increment* for ans_i, denoted by $inc(ans_i)$.

The increment approximation for a forest logging trace is the set of best-fit increments for all answers: $\{inc(ans) \mid ans \in answers(as_i)\}$. They are likely to be the functors applied repeatedly to derive more and more answers and thus causing the computation to not terminate. This analysis can help the programmer identify the rules of the program, which are actually being fired in these derivation cycles.

Theorem 5 (Completeness). *Consider a query to a program all of whose predicates are tabled and assume that the inference engine supports subgoal abstraction, as before. If query evaluation does not terminate, then the increment approximation of the complete infinite trace contains all the functors that repeat themselves in the answer segments and for which the number of repetitions grows to infinity.* □

A proof of this theorem and an algorithm for computing increment approximations can be found in [7].

Example 7. The first column of Table 2 shows the first three answer segments of the forest logging trace for Example 3. Column 2 is the answer contained in each answer-generating log entry, and column 3 contains the best-fit increment for each answer. The increment approximation of the trace is the union of all answer increments: $\{[g], [f, g]\}$. □

[2] A *safe* knowledge base is one in which all base facts are ground and rules have no unsafe variables, i.e., all variables in the rule heads also occur in the rule bodies.

Table 2. Answer Increments for Example 3

Answer Segments	Answers	Best-fit Increments
`na([g(a)],q(_v0),6)`	$g(a)$	
`na([f(b)],p(_v0),8)`	$f(b)$	
`na([f(g(a))],p(_v0),10)`	$f(g(a))$	
`na([g(f(b))],q(_v0),12)`	$g(f(b))$	$[g]$
`na([g(f(g(a)))],q(_v0),14)`	$g(f(g(a)))$	$[g]$
`na([f(g(f(b)))],p(_v0),16)`	$f(g(f(b)))$	$[f, g]$
`na([f(g(f(g(a))))],p(_v0),18)`	$f(g(f(g(a))))$	$[f, g]$
`na([g(f(g(f(b))))],q(_v0),20)`	$g(f(g(f(b))))$	$[g]$
`na([g(f(g(f(g(a)))))],q(_v0),22)`	$g(f(g(f(g(a)))))$	$[g]$
`na([f(g(f(g(f(b)))))],p(_v0),24)`	$f(g(f(g(f(b)))))$	$[f, g]$
`na([f(g(f(g(f(g(a))))))],p(_v0),26)`	$f(g(f(g(f(g(a))))))$	$[f, g]$

3.5 Computational Complexity

Consider a forest logging trace and let N_{tc}, N_{ar} and N_{na} be the numbers of *tc*-facts, *ar*-facts and *na*-facts in the trace, respectively. We assume that the number of unfinished calls in the trace is $N_{unfinished}$ and the size of optimal child-parent pattern found in answer flow analysis is N_{cpp}. $N_{unfinished}$ is typically much smaller than N_{tc} since many calls would have been completely evaluated before non-termination occurs. Similarly, N_{cpp} is usually much smaller than N_{ar}. The complexity of different operations in `Terminyzer` is summarized in Table 3, and detailed analysis can be found in [7].

Table 3. Complexity Summary

Operation	Complexity
Computing uncompleted calls	$O(N_{tc})$
Constructing uncompleted-call CPG	$O(N_{unfinished})$
Computing all uncompleted-call paths from *root*	NP
Computing a culprit uncompleted-call path	$O(N_{unfinished})$
Computing optimal child-parent pattern	$O(N_{ar}^2)$
Constructing flow-pattern CPG	$O(N_{cpp})$
Computing all flow-pattern paths	NP
Computing all flow-pattern loops	NP
Computing best-fit increment for one answer	$O(N_{na})$
Computing increment approximation for a trace	$O(N_{na}^2)$

4 Experiments

We implemented a prototype of `Terminyzer` in XSB-Prolog, which works both for the XSB and $\mathcal{F}$LORA-*2* deductive systems. We tested the analyzer using several programs and some of the results are shared in this section. In the experiments, we set XSB to abort queries after the answer depth reached the depth of 30. All tests were performed on a dual core 2.4GHz Lenovo X200 with 3 gigabytes of main memory running Ubuntu 11.04 with Linux kernel 2.6.38.

Test Programs. Here we present the results for four test cases: T_1, T_2, T_3 and T_4 of which only T_1 terminates. Test T_1 evaluates the query `?- path(1,Y)` against the transitive closure program, as in Example 1. T_2 evaluates the query `?- p(X)` against the program in Example 3. T_3 computes the query `?- p(X)` for the following program:

```
:- table p/1.
p(a).  p(b).
p(f(X)) :- p(X).
```

Unlike the other three cases, T_4 is a very large program, which was derived from a $\mathcal{F}$LORA-*2* program used in the SILK Project[3] that has 4,774 rules and 919 facts. The corresponding XSB program (after the $\mathcal{F}$LORA-*2*-to-XSB translation) is estimated to have over 1,000 facts and over 5,500 rules. Execution produces a log trace in excess of 500 megabytes with 2,749,822 log entries.[4]

Test Results. `Terminyzer` produced expected results in all the test cases. For T_1, it produced an unfinished-call CPG $(\{root\}, \emptyset)$ and an empty answer-flow CPG. No answer increments were produced. For T_2, `Terminyzer` produced the unfinished-call graph shown in Figure 2(B) of Example 3 and the unfinished-call loop as in Example 4. It also constructed the answer-flow graph and found the answer-flow loop given in Example 6. Furthermore, it found the increment $\{[g], [f, g]\}$ as in Example 7.

For T_3, `Terminyzer` computed two unfinished calls: $unfinished\_call(p(\_v0), root, 0)$ and $unfinished\_call(p(\_v0), p(\_v0), 3)$. It also constructed the unfinished-call CPG $(\{root, p(\_v0)\}, \{(root, p(\_v0)), (p(\_v0), p(\_v0))\})$ and answer-flow CPG $(\{p(\_v0)\}, \{(p(\_v0), p(\_v0))\})$. The unfinished-call loop and answer-flow loop are both $[p(\_v0)]$. `Terminyzer` also correctly found the increment $\{[f]\}$.

For T_4, the unfinished-call CPG has eight nodes and twelve edges, and it contains five unfinished-call loops. Its answer-flow CPG has three nodes and six edges with four answer-flow loops. `Terminyzer` also identified an increment of length 8.

Computation Times. Call-sequence, answer-flow and functor-pattern analyzers all took less than one second for T_1, T_2, and T_3. For T_4, the call-sequence analysis finished within 1 second, while answer-flow and functor-pattern analysis together took 23 seconds.

One optimization would be to split forest logging traces into multiple files for different analyzers, since different analyzers largely make use of different entries in the trace: call sequence analysis uses only the *tc*-facts and *cmp*-facts; answer-flow analysis needs *ar*-facts; while functor-pattern analysis uses *ar*-facts and *na*-facts. Entries that are irrelevant for a particular analysis can be deleted

[3] `http://silk.semwebcentral.org/`

[4] We also tested other, smaller, but still fairly large real programs from the SILK project with similarly positive results.

thereby significantly reducing the size of the data set that we need to deal with. This optimization is implemented in `Terminyzer` as a pre-processor.

5 Related Work and Conclusion

There have been extensive studies on termination analysis for logic programs [14,18,8,1,10,11,13] and much less work on *non*-termination analysis [4,9,12,20,19]. There are two major points that differentiate our work. First, the termination and non-termination problems discussed in most previous works are non-issues in our framework: those problems stem from the incompleteness of the Prolog inference mechanism and therefore do not apply to our case. Second, `Terminyzer` utilizes traces to *help* the programmer *debug* his programs *without syntactic restrictions* . All other approaches perform static or dynamic analysis in order to *prove* termination and non-termination properties of *restricted* classes logic programs.

Our immediate future plans include adding program *justification*, When justification is included, `Terminyzer` will not only be able to produce the sequences of calls that might be causing non-termination, but also sequences of rules that were fired in SLG resolution for each such call. We are also working on auto-repair of non-terminating behaviors once non-termination is detected.

Acknowledgments. We would like to thank Terry Swift for enabling this work by implementing subgoal abstraction and forest logging in XSB. We are also thankful to the referees for their helpful critique.

References

1. Bruynooghe, M., Codish, M., Gallagher, J.P., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms. ACM Trans. Program. Lang. Syst. 29 (April 2007)
2. Chen, W., Kifer, M., Warren, D.S.: HiLog: A foundation for higher-order logic programming. J. Log. Program. 15(3), 187–230 (1993)
3. Costa, V.S., Damas, L., Rocha, R.: The YAP prolog system. CoRR abs/1102.3896 (2011)
4. Decorte, S., De Schreye, D., Leuschel, M., Martens, B., Sagonas, K.: Termination analysis for tabled logic programming. In: Fuchs, N.E. (ed.) LOPSTR 1997. LNCS, vol. 1463, pp. 111–127. Springer, Heidelberg (1998)
5. Hermenegildo, M.V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J.F., Puebla, G.: An overview of ciao and its design philosophy. TPLP 12(1-2), 219–252 (2012)
6. Liang, S.: Non-termination analysis and cost-based query optimization of logic programs. In: Krötzsch, M., Straccia, U. (eds.) RR 2012. LNCS, vol. 7497, pp. 284–290. Springer, Heidelberg (2012)
7. Liang, S., Kifer, M.: Terminyzer: An automatic non-termination analyzer for large logic programs. In: Technical Report (2012)

8. Lindenstrauss, N., Sagiv, Y., Serebrenik, A.: Proving termination for logic programs by the query-mapping pairs approach. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 453–498. Springer, Heidelberg (2004)
9. Neumerkel, U., Mesnard, F.: Localizing and explaining reasons for non-terminating logic programs with failure-slices. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 328–342. Springer, Heidelberg (1999)
10. Nguyen, M.T., De Schreye, D.: Polytool: proving termination automatically based on polynomial interpretations. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 210–218. Springer, Heidelberg (2007)
11. Nguyen, M.T., Giesl, J., Schneider-Kamp, P., De Schreye, D.: Termination analysis of logic programs based on dependency graphs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 8–22. Springer, Heidelberg (2008)
12. Payet, E., Mesnard, F.: Nontermination inference of logic programs. ACM Trans. Program. Lang. Syst. 28, 256–289 (2006)
13. Schneider-kamp, P., Giesl, J., Ströder, T., Serebrenik, A., Thiemann, R.: Automated termination analysis for logic programs with cut*. Theory Pract. Log. Program. 10, 365–381 (2010)
14. Schreye, D.D., Decorte, S.: Termination of logic programs: The never-ending story. J. Log. Program. 19/20, 199–260 (1994)
15. Sipser, M.: Introduction to the Theory of Computation. Thomson Publishing (1996)
16. Swift, T., Warren, D.S., Sagonas, K., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Marques, R.F., Saha, D., Dawson, S., Kifer, M.: The XSB system, version 3.3.x.: Programmer's manual (2012), `http://xsb.sourceforge.net`
17. Swift, T., Warren, D.S.: Xsb: Extending prolog with tabled logic programming. CoRR abs/1012.5123 (2010)
18. Verbaeten, S., Schreye, D.D., Sagonas, K.: Termination proofs for logic programs with tabling. ACM Trans. Comput. Logic 2(1), 57–92 (2001)
19. Voets, D., De Schreye, D.: Non-termination analysis of logic programs using types. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 133–148. Springer, Heidelberg (2011)
20. Voets, D., De Schreye, D.: A new approach to non-termination analysis of logic programs. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 220–234. Springer, Heidelberg (2009)
21. Zhou, N.F.: The language features and architecture of b-prolog. TPLP 12(1-2), 189–218 (2012)

Integrative Functional Statistics in Logic Programming

Nicos Angelopoulos[1,2], Vítor Santos Costa[3,4], João Azevedo[5], Jan Wielemaker[6], Rui Camacho[5], and Lodewyk Wessels[1,2]

[1] Bioinformatics and Statistics, Netherlands Cancer Institute, Amsterdam, Netherlands
n.angelopoulos@nki.nl
[2] The Netherlands Consortium for Systems Biology (NCSB)
[3] CRACS-INESC Porto LA, Universidade do Porto
[4] DCC-FCUP, Universidade do Porto
Rua do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
vsc@dcc.fe.up.pt
[5] LIAAD & DEI & Faculdade de Engenharia, Universidade do Porto, Portugal
[6] Vrije Universiteit Amsterdam, Netherlands

Abstract. We present *r..eal*, a library that integrates the R statistical environment with Prolog. Due to *R*'s functional programming affinity the interface introduced has a minimalistic feel. Programs utilising the library syntax are elegant and succinct with intuitive semantics and clear integration. In effect, the library enhances logic programming with the ability to tap into the vast wealth of statistical and probabilistic reasoning available in *R*. The software is a useful addition to the efforts towards the integration of statistical reasoning and knowledge representation within an AI context. Furthermore it can be used to open up new application areas for logic programming and AI techniques such as bioinformatics, computational biology, text mining, psychology and neuro sciences, where *R* has particularly strong presence.

1 Introduction

Logic programming provides a powerful framework for reasoning with complex, structured data and is an important vehicle for AI research. The Prolog language is a popular example of logic programming that provides a query driven inference mechanism. Prolog has been shown to be useful in diverse AI application domains, including machine learning, natural language, data-base interfacing, web services, and program analysis. Often, Prolog applications require computing aggregate properties of data. Common operations, such as computing the mean or standard deviation, can be easily programmed in Prolog. More complex operations, such as clustering, pattern extraction or likelihood computations can be hard to implement efficiently.

The *R* environment [15] is an open source software package for statistical data analysis. *R* is widely used by the statistical and data mining communities, with major applications in areas such as bioinformatics. The *R* environment provides a set of effective tools for data storage and manipulation, namely of arrays, and it implements a well developed programming language, *S* [5]. Although *S* is not a pure declarative language, it contains a strong functional programming component. *R* also has an excellent packaging and distribution system through which a multitude of researchers and programmers make their code available to the community. The comprehensive *R* archive network (CRAN, `http://cran.r-project.org/`) contains a vast selection of contributed code that deals with the full gamut of statistical inference and data analysis. Examples include several implementations of the pagerank algorithm [8], machine learning tools such as support vector machines [10] and several clustering tools [13].

Prolog has been the main practical vehicle of logic programming with a number of open source implementations available to the community. *YAP* [7] and *SWI* [21] are two such systems. The first is widely accepted as one of the fastest open source Prolog implementation in, among other areas, machine learning and particularly in inductive logic programming (ILP) [18]. Machine learning is one application where it is natural to interface a Prolog system to *R*, namely to extend the ILP paradigm with regression capabilities [1]. The *SWI* Prolog system is the most widely used open source Prolog with many educational, research and industrial installations. It is well regarded for its stability and extensive palette of libraries. The software presented here has been developed to run on both of these Prolog systems, and is made available as open source software in the hope that it can be widely adopted and become a standard that enhances Prolog's capabilities.

A further motivation for integrating *R* with logic programming stems from the observation that traditionally, work on logic programming has focused in representing crisp knowledge. More recent work on the interplay between knowledge representation and statistical inference has attracted substantial interest in recent years. Work in this area includes the *PRISM* system and its EM-based parameter algorithm [16], Stochastic logic programs with an MCMC structure learning system [2] and the FAM algorithm [9], the ProbLog language and system [12] with a variety of learning algorithms and CLP($\mathcal{BN}$) [6], with EM learning and an interface to *Aleph* [18]. The interface to *R* allows the integrated statistical-logic inference systems access to a wide range of tools from random number generators to sophisticated algorithms for probabilistic inference.

Our work, the *r..eal* library, overcomes significant shortcomings present in earlier attempts to interface Prolog to *R* [1,4]. The first approach, *YapR*, used the *C* interface to pass *R* commands as sequences of characters with little conversion. In the second approach, *r_session*, the interface provided an expression based communication via an independent *R* process to which the operating system channelled I/O from Prolog. This approach was more flexible, but inefficient and hard to maintain across operating systems. The new approach, *r..eal*, introduces a completely new design that provides a tightly integrated interface to *R* for the Prolog programmer. In our approach *R* is invoked as a shared operating system library while the communication of large data between Prolog and *R* is facilitated by *C* code utilising the *C*-interfaces for the two systems.

We argue that critical to Prolog's success as a vehicle for AI research is its ability to address statistical aspects of knowledge representation and reasoning. Consider one domain which is currently experiencing a rapid expansion: computational biology. In this domain, vast volumes of data need to be interpreted and resulting knowledge to be represented. *R* packages such as *Bioconductor* [11] are among the most successful tools in this area, but they lack the knowledge representation strengths of Prolog. Thus, by combining logic programming with the extensive statistical functionality of *R*, we hope to contribute to progress in this field while engaging more of the logic programming community in this area of research.

The rest of the paper is organised as follows. Section 2 presents the developed interface. Data representation in *R* and Prolog and the translation process is described in Section 3. Section 4 shows some illustrative examples and the conclusions are summarised in Section 5.

2 Interface

R..eal enables the communication between the Prolog system and *R*. The *R* environment executes as an operating system library: from the Prolog point of view, *R* is just another set of functions; from the *R* point of view, Prolog is the top-level. The user interface is designed to satisfy the following requirements:

- *Minimality:* ideally, most interactions should be performed through a small number of predicates.
- *R Flavour:* using the interface should be as close as possible to the standard usage of *R*. It should feel as if we are writing *R* code. To do so, most common *R* constructs should just work.
- *Prolog Flavour:* the interface should not require the user program to construct a sequence of characters to be interpreted by *R*. Instead, it should be about Prolog terms that are constructed and manipulated by Prolog code.

Arguably, the two last goals are incompatible, given the conceptual and syntactic differences between Prolog and *R*. *R..eal* tries to be as close to *R* as possible, but respecting the observation that ultimately one has to construct a valid Prolog program.

The library leaves the management of *R* variables to the programmer. On backtracking there is no removal of variables from the *R* environment. In practice, this is rarely a limitation, particularly since *R* variables can be destructively assigned new values. In our experience, the strengths of Prolog search through solutions spaces, merge well with a sequential application of *R* functions that can provide statistical computations.

2.1 Access Predicates

The *R* language uses `<-` as the assignment operator. In order to be as close to possible to *R* syntax, *r..eal* uses `<-/1` and `<-/2` to channel the bulk of the interactions between Prolog and *R*. The predicate names are defined as prefix and infix operators, respectively. The `<-/1` predicate sends an *R* expression, represented as a ground Prolog term, to *R*. The `<-/2` operator facilitates bi-directional communication. If the left-hand side is a free variable, the library assumes that we are passing data from *R* to Prolog. If the left-hand side is bound, *r..eal* assumes that we are passing data or function calls to *R*. The library implements two communication mechanisms:

- arbitrary *R* expressions of function calls which possibly embed data items within their arguments, are transformed from Prolog terms to strings and passed to *R* for native parsing

- Prolog lists and *R* vectors are passed by *r..eal* through *C* code that understands how Prolog and *R* represent data;

More concretely, the calling modes for `<-/2` are:

```
+Rvar    <-  +PLvalue
-PLvar   <-  +Rvar
-PLvar   <-  +Rexpr
+Rexpr1  <-  +Rexpr2
```

In the first, top-most mode, the *C* interface is employed to transfer Prolog data value(s), `PLvalue`, to an *R* variable identified by `Rvar`. In the second mode, *r..eal* instantiates the Prolog variable `PLvar` to the contents of the *R* variable `Rvar`. In the second mode `Rexpr` is evaluated in *R* and its result is unified to Prolog variable `PLvar`. In the current implementation this is done by first assigning the result to a hidden *R* variable and then using the second mode to copy this onto `PLvar`. In the last mode, *r..eal* will pass `Rexpr1 <- Rexpr2` to *R* subject to the syntactic conversions described in the next section. *R..eal* will automatically distinguish between the four modes. A variable in the left side of the operator is taken to be a Prolog variable, an atom is recognised as an *R* variable (`Rvar` above) and a ground term is considered to be an *R* expression. On the right side, a list or a number are taken as Prolog data, an atom corresponding to a known *R* variable is recognised as such and all other terms are *R* expressions.

In the following example a list of 6 Prolog integers is passed to the *R* variable `v` and then passed to Prolog variable `V`.

```
?-  v <- [0,1,1,2,3,5],
    V <- v.

V = [0, 1, 1, 2, 3, 5].
```

In the arity 1 version of the assignment predicate, if the argument can be interpreted as a known *R* variable then it is printed using the *R* function call `print()`. The following example prints the contents of an *R* variable (`v`) that has been passed a list of Prolog integers.

```
?-  v <- [0,1,1,2,3,5],
    <- v.

[1] 0 1 1 2 3 5
```

When *r..eal* cannot establish that the argument of `<-/1` is an *R* variable, it passes the argument to *R* as an expression right after all syntactic transformations have been completed. This allows for calling of functions to which the return value is of no interest to the user. For instance the value of the plotting function is often ignored. The following example uses *R*'s `plot()` function to plot 3 points with x-coordinates $[1, 2, 3]$ and y-coordinates $[4, 5, 6]$. The plot appears on *R*'s default plot display.

```
<-  plot( [1,2,3], [4,5,6] ).
```

3 Data Representation in R

R recognises several types of objects:

– Floating point numbers, integers, Boolean and ascii values (character strings) provide the base types.
– Lists or vectors are the main forms of serialised compound objects.
– Arrays are multi-dimensional compound objects with two dimensional arrays treated as special arrays called matrices.
– *R* supports several useful data-types: dotted-pairs are used to represent lists; the : operator is supported for ranges, and `NULL` objects represent uninitialised *R* objects.
– Programs can be constructed by using *symbols*, functions or closures, and environments.

Regarding base types, there are matches between floating point and integers in *R* and Prolog. Boolean values can be matched to `true` and `false` atoms. Character strings are traditionally represented by Prolog as lists of character codes. These principles correspond to the following rules:

```
Prolog      ---  R
integer     <->  integer
float       <->  double
atom        <->  char
char         ->  char
true/false <->  logical
```

The three other major types supported by the interface are symbols, vectors and matrices. Symbols are *R* identifiers used for variable and function names. They naturally map to Prolog atoms and they are contextually distinguished from chars. Compound objects are described in detail next.

3.1 Vectors and Matrices

Vectors are a key generic data type in *R*. It is important to make two observations on the nature of vectors in *R*. First, that *R* vectors are typed and second that they have attributes. *R* has six basic vector types: logical, integer, real, complex, string (or character) and raw. The other major data types in *R* include lists, expressions and functions. As an example, the *R* variable `v`, defined by

```
?-  v <- as..integer(c(1,2,3)).
```

is of type integer vector and its contents are the values $1, 2$ and 3. Note that `c()` is a generic method in *R*. The default function of this method is to combine its arguments into a vector. A vector naturally translates to a list in Prolog. Multi-dimensional arrays are mapped to lists of lists. This principle works both ways: Prolog lists are mapped to vectors, and lists of lists to matrices (which are 2 dimensional arrays in *R* parlance). *R..eal* provides two main ways to pass Prolog data to *R*. The more efficient method is by using the *C* interface while the alternative method constructs a string representation of an *R* command. The former method is accessible to the user via the mode of `<-/2` in which Prolog data is passed to an *R* variable. Goals have the following form, where `PLvalue` is the Prolog data and `Rvar` is the *R* variable.

```
        +Rvar <- +PLvalue
```

This mode is implemented in *C* and transfers via *C* data from Prolog to *R*. The type of values of the vector or matrix is taken to be the type of the first data element of the `PLvalue`. An example of passing a list of the integers between 1 and 100 to an *R* variable (`i`), printing the first ten elements through *R* and then passing the vector back to Prolog after adding 1 to each number follows:

```
    ?-  findall( I, between(1,100,I), Is ),
         i <- Is,
         <- i^[1:10],                % prints via R
         Js <- as..integer(i+1).

    [1]  1  2  3  4  5  6  7  8  9 10
    Is = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
    Js = [2, 3, 4, 5, 6, 7, 8, 9, 10|...].
```

3.2 Object Types

When passing Prolog objects to *R*, *r..eal* attempts to build the *R* structure by extrapolating the object type from the type of the first element in the Prolog structure. If later on this breaks down, the structure is rebuilt if the type that introduced the failure can be used for the overall data structure.

For example, the Prolog list in the following query contains a float value in its third position. As Prolog is untyped, we do not have this information when the list starts being transferred across. Instead, at the start of passing the list through, the first element is inspected and the list is assumed to contain integers. At the third element, upon encountering a float value, the work done so far is scrapped and the more general type is used to translate the list to a vector of float values.

```
    ?-  r <- [1,2,3.2,4],
        <- r,
        R <- r.

    [1] 1.0 2.0 3.2 4.0
    R = [1.0, 2.0, 3.2, 4.0].
```

3.3 The Expression Mechanism

Data appearing in an arbitrary *R* expression is parsed and placed into a string that will then be passed from Prolog to *R* for evaluation. For instance, in the following example the `c()` combinator function is used to combine 5 values into an *R* vector before printing it and then pasting all vector elements to a single value vector (`s`). For illustration purposes we also include a goal that combines the two function calls (assignment to *R* variable `t`).

```
    ?-  state <- c("tas","sa","qld","nsw"),
        <- state,
        s <-paste(state,collapse="+"),
        t <-paste(c("tas","sa","qld","nsw"),collapse="+"),
        <- s,
        <- t.

    [1] "tas" "sa"  "qld" "nsw"
    [1] "tas+sa+qld+nsw"
    [1] "tas+sa+qld+nsw"
```

The implementation of *r..eal* recognises that the expression to be assigned to *R* variable *t* is not a single Prolog data term but a number of *R* function calls, so it transforms this expression into a string containing an *R* expression. Note that when using this interface it is convenient to represent *R* chars by Prolog list of codes, as in the above example.

Passing long objects through the expression mechanism is both inefficient and can easily lead to buffer limitations as it is only intended as a mechanism for passing function calls on existing *R* objects. *R..eal* circumvents both these limitations by automatically detecting Prolog lists and `c()` terms and passing them via a *hidden R* variable which is then substituted in the call passed for evaluation to *R*. The temporary name of the hidden variable is selected so as not to clash with the current *R* name-space.

For instance, the following code generates a list of $50,000$ elements and computes the mean of its elements via a call to *R* through the expression mechanism. Without the use of hidden variables this call would generate a resource error and even shorter lists would take much longer to transfer. The example code that follows was executed on SWI-Prolog 6.3.0 on a Linux 11.10 desktop having a dual core 3.16 GHz processor.

```
    ?-  findall(I,  between(1,50000,I), Is),
        time( A <- mean(Is) ).

    % 181 inferences,0.002 CPU in 0.002 seconds
                                        (100% CPU,75597 Lips)
Is = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
A = 25000.5.
```

In the above calls, `A <- mean(Is)` becomes `t <- Is, A <- mean(t).`

3.4 Syntactic Issues

There are syntactic conventions in *R* which result in non-parsable Prolog code. Notably function and variable names are allowed to contain dots, square brackets are used to access parts of vectors and arrays, and functions are allowed empty argument tuples.

We have introduced syntax which allows for easy translation between Prolog and *R*. Prolog constructs are converted by the library as follows:

– *R* code often uses the '`.`' symbol with function and variable names. As this syntax conflicts with standard Prolog usage, *r..eal* allows the use of the operator '`..`', e.g.:

```
as..integer(c(1,2)) => as.integer(c(1,2))
```

The library's name is a word play on the '`..`' operator.

– *R* allows matrix subscripts. In the style of BProlog [22], *r..eal* uses the '`^`' operator:

```
a^[2] => a[2]
```

– *R* allows ranges over subscripts, say `a[,,2]` which in *R* is a way of to refer to all the values of the first and second dimension of `a`. *R..eal* uses `*` for this purpose:

```
a^[*,*,2] => a[,,2]
```

Note that *r..eal* follows *R* conventions to access arrays.

– We map the '`$`' *R* operator to a Prolog library operator (`op(400,yfx,$)`). In *R*, `$` is one of the possible ways in which parts of vectors, matrices, arrays and lists can be extracted or replaced. In most contexts there is no ambiguity so the operator can be used freely, however in some situations it might be necessary to quote.

```
a$val => a$val   or   'a$val' => a$val
```

– *R..eal* uses `(.)` to denote *R* functions with zero arity:

```
dev..off(.) => dev.off()
```

– The *R* `NULL` value is coded as the empty list.

– Simple *R* functions can be coded by using the Prolog implication operator '`:-`':

```
(f(x) :-  (...)) => f(x) (...)
```

This is only advised for very small functions, and does not support conditionals yet.

– As mentioned previously, lists of lists are converted to matrices. In contrast to the flexibility of *R*, all levels of the lists must have the same length.

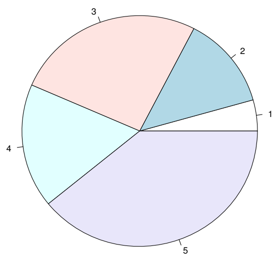
– Prolog represents character strings as lists of integers. It is thus impossible to distinguish strings from genuine lists of integers appearing in arbitrary *R* expressions. We define '`+`' as a prefix operator to identify strings.

```
source(+"String")  =>  source("String")
```

– Some *R* operators should be quoted in Prolog:

```
a '%*%' b  => a %*% b
```

The majority of *R* operators can be used unquoted as they are defined as infix operators and present no issues. Finally, expressions that *r..eal* cannot translate can always be passed as Prolog atoms or strings.



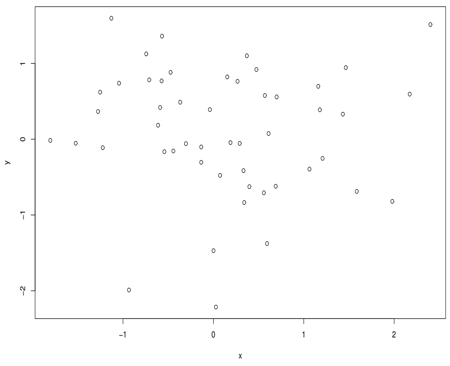


Fig. 1. Left: pie-chart example of a Prolog list of integers. Right: plotting two sequences of 50 random numbers as X and Y coordinates.

4 Applications

4.1 Plot Drawing

Visualising data is a particular strength of *R*, whereas Prolog systems traditionally have only limited access to graphics. The *r..eal* interface enables access to the extensive facilities of *R*. Simple plots such as scatter plots, histograms, box plots and pie charts can be easily drawn for Prolog data objects. In Figure 1 a pie chart and a scatter plot are shown. In the first example a list of integers is passed and plotted as a pie-chart, where each integer indicates the relative area of each slice. The following is the code for drawing the pie-chart shown in the LHS of Fig. 1.

```
?-  cars <- [1, 3, 6, 4, 9],
    <- pie(cars).
```

The next example, also from Fig. 1, shows how to create a plot of 50 random samples whose coordinates have been drawn from a normal distribution (`rnorm()`). The coordinates are stored in *R* variables `x` and `y` before being plotted on a new plotting window created with the `x11()` function. Over twenty different probability distributions are present in *R*, with more available in add-on packages.

```
?-  <- set..seed(1),
    y <- rnorm(50),
    x <- rnorm(y),
    <- x11(width=5,height=3.5),
    <- plot(x,y).
```

A third plotting example is shown in Figure 2. In this case, the nested outer product of two vectors defined implicitly using the column notation (`0 : 9`) is computed. The results are then tabled and plotted. Labels are passed to the plot via variables instantiated to character strings. The generating code is as follows:

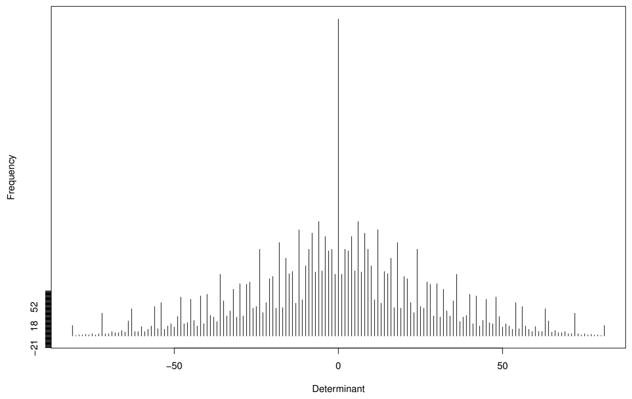


Fig. 2. Tabled nested outer product of $0 : 9$ to itself

```
?-  d <- outer( 0:9, 0:9 ),
    fr <- table(outer(d, d, +"-")),
    Xl = "Determinant", Yl = "Frequency",
    fr..names <- as..numeric(names(fr)),
    <- plot(fr..names,fr,type=+"h",xlab=+Xl,ylab=+Yl).
```

4.2 Interacting with Datasets

Interfacing to *R* allows access to a large variety of data formats. As an example, consider the comma-separated values (`csv`) format file `trees91.csv`:

```
C,N,CHBR,REP,LFBM,STBM,RTBM,LFNCC,STNCC,RTCACC,LFKCC,STKCC,...
1,1,CL6,1,0.43,0.13,0.29,1.84,0.4,0.96,0.13,0.06,0.23,0.3,...
1,1,CL7,1,0.4,0.15,0.25,1.82,0.37,0.95,0.18,0.06,0.22,0.22,...
1,2,A1,9,0.45,0.2,0.21,1.54,0.96,0.69,0.16,0.08,0.3,0.35,...
...
```

The first line contains headers, and the remaining lines contain data in a tabular format, separated by commas. Reading the file into an *R* variable (`tree`) is done by simply calling:

```
?-  tree <- read..csv(file="trees91.csv",
                              sep=",",head='TRUE').
```

For instance, to get the column names in a Prolog list we can do:

```
?-  X <- names(tree).

X = ['C','N,'CHBR','REP'|...].
```

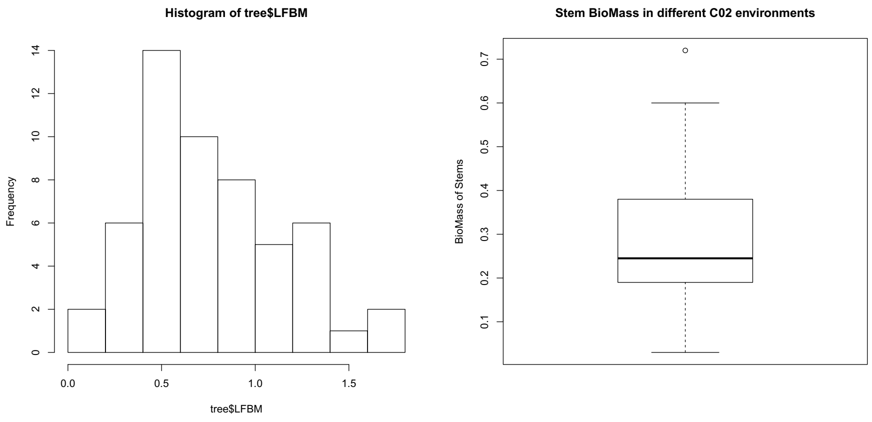


Fig. 3. Reading csv data. Left: histogram of `LFBM` column. Right: box plot of `LFBM` column.

The `$` operator is used to access columns of the read table:

```
?-  X <- tree$'LFBM'.

X = [0.43,0.4,0.45,0.82,0.52,1.32,0.9,1.18,0.48|...].
```

It is straightforward to obtain and plot a histogram of a specific column (LHS, Fig. 3):

```
?-  X <- hist(tree$'LFBM').

X = [breaks=[0.0, 0.2, 0.4, ...], ... ].
```

This will both fill X with the histogram and plot it in the graphical interface (RHS, Fig. 3). Plotting can be avoided when passing the FALSE value to the argument `plot`, while the histogram can also be plotted without any value being explicitly returned by using `<-/1`:

```
?-  X <- hist(tree$'LFBM', plot = 'FALSE').

X = [breaks=[0.0, 0.2, 0.4, ...], ... ].

?-  <- hist(tree$'LFBM').
```

The same plot can be saved in a PDF file. The function `pdf()` opens a new graphics device with output to the named PDF file, while the *R* function `dev.off()` closes the graphics device that was opened last.

```
?-  <- pdf(+"plot.pdf"),
    X <- hist(tree$'LFBM'),
    <- dev..off(.).

X = [breaks=[0.0, 0.2, 0.4, ...], ... ].
```

The data can be displayed in a variety of different formats, for example as box plots (RHS, Fig.3).

```
?- Main="Stem BioMass in different C02 environments",
   Y = "BioMass of Stems",
   <- boxplot(tree$'STBM', main=+Main, ylab=+Y).
```

4.3 Pagerank on Prolog Programs

We have so far seen how Prolog can command *R* and pass data to it. We can further observe that Prolog programs are term structures themselves, suggesting that we might want to apply a variety of well known statistical algorithms to the analysis of Prolog programs. The next example shows an application where we take advantage of the respective strengths of Prolog and *R*. The goal is to find the most important, or cross-referenced, procedure in a Prolog program by using a graph algorithm on a network representing the call dependencies of the program under investigation. We use the popular pagerank algorithm [14] and its implementation in *R*'s *igraph* package [8]. We apply our analysis on the source code of the ILP program *Aleph* [18].

The first building block of our program visits the Prolog source and constructs a graph where the nodes are the predicates used by *Aleph*. We define procedure `parse/2` to collect all edges in a source file:

```
parse(File, Nodes) :-
   open(File, read, S),
   findall(Node, clause_to_nodes(S, Node), Nodes),
   close(S).
```

The program scans every clause in the file. For each clause, it first maps the head and every sub-goal in the body to an integer corresponding to its defining predicate. Then, it creates an edge between every sub-goal and the head. As an example, in the following clause

```
subtract([E|T], D, R) :-
    memberchk(E, D), !,
    subtract(T, D, R).
```

the program will first map `subtract/3` to 0 and `memberhck/2` to 1 and then generate the graph $\{1 \mapsto 0, 0 \mapsto 0\}$. As *R*'s `graph()` function prefers to receive the graph as a list of nodes we make `clause_to_nodes/2` succeed four times with answers $Node = 1$, $Node = 0$, $Node = 0$, and $Node = 0$, so that two consecutive solutions represent an edge. Solutions are then captured by `findall/3` as a list that is passed on to the *R* environment and the `graph` function.

We define the `pagerank/1` procedure to find the maximum element of the graph nodes in a file as computed by `page.rank()`.

```
pagerank(File,nav(Name,Arity,Value)) :-
   parse(File,Graph),
   g <- graph(Graph),
```

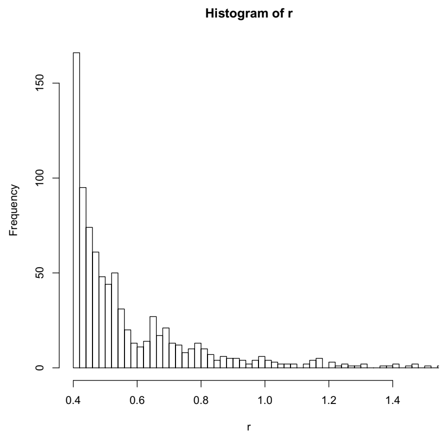


Fig. 4. Histogram of pagerank scores for *Aleph* predicates

```
    r <- page..rank(g),
    Scores <- r$vector,
    max_element(Scores, Name, Arity, Value).
```

This predicate obtains the list of edges, and calls `graph()` to create a weighted graph `g`, where the weight is the number of repeated occurrences. It then computes the page rank scores into the *R* object `r` and reads the vector component of the object to list $Scores$. The `max_element/3` procedure simply extracts the predicate with highest pagerank.

Applying the program to ILP system *Aleph* generates a graph with 968 nodes and 7296 edges. The highest score in the graph is for `!`, which is unsurprising. If we remove all built-in predicates the highest score is for `$aleph_global/2`. We can also reverse the graph. In this case the highest score is for `reduce/0`. Figure 4 shows a histogram of pagerank scores for the predicates in *Aleph*.

4.4 Search and Visualisation

R and Prolog are complementary in that the former has strong presence in data analysis and visualisation while the latter has strengths in knowledge representation and search based reasoning. In order to underline this and point at computational biology and bioinformatics as important areas of applications, we employ *r..eal* as a bridge between a search algorithm implemented in Prolog and visualisation component via the *RCytoscape* [17] *Bioconductor* package.

The objective is to build a network of interactions between genes that encode proteins that are involved in cell motility. Direct edges, representing interactions within this set of genes (the *adhesome library*), are extracted from the Kyoto Encyclopedia of Genes and Genomes (KEGG). We then employ a breadth first algorithm to join disconnected adhesome genes to the main network by finding one of the shortest paths involving the

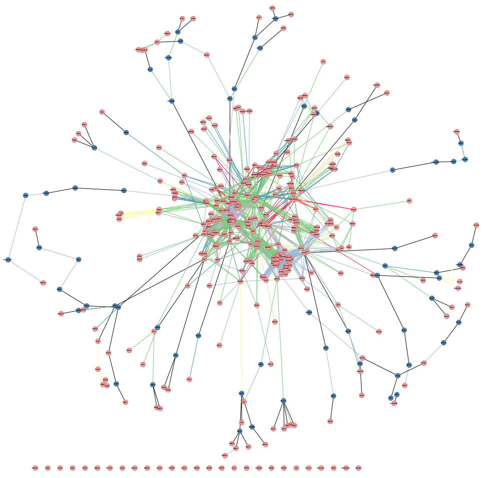


Fig. 5. KEGG interactions for a subset of the members of an adhesome library. Nodes are proteins and edges denote interactions. Blue nodes are connecting proteins that do not appear in the library. Edges are coloured as per type of interaction.

minimum number of intermediary genes that are not in the library. The results can then be displayed via the *RCytoscape* package which interfaces *R* to the *Cytoscape* biological network visualiser.

Figure 5 shows the graph of the adhesome library as searched by Prolog and displayed by *RCytoscape* via calls to *r..eal*. This example utilises an in-house library that uses *R* and its bridge to *Cytoscape* (*RCytoscapse*) to display arbitrary Prolog graphs (as those managed with the *u*graph library). We hope to publish this soon, inclusive of the code for this example. The connection established with *RCytoscape* is bi-directional. The user can use all facilities in *Cytoscape*, such as selecting nodes or edges. Lists of such selections can be queried via *r..eal* which can be a starting point for further analysis and search within Prolog. A more general discussion on Prolog and *r..eal* for bioinformatics can be found in [3].

5 Conclusions

The library presented here achieves a tight integration of the *R* statistical software system with two open source Prolog implementations. Our designing principles have been those of simplicity and transparency across the systems. This has been accomplished by (a) keeping to a minimum the transformations the user needs to be aware of, and by (b) providing intuitive, mnemonic syntax to the inconsistencies between the two languages. As a result, *r..eal* programs are clear and easy to follow. The functional inheritance of *R* corresponds well with the logical underpinning of Prolog. *R..eal* provides a productive environment for building highly effective pipelines and interactive, query-based data exploration.

Interfacing the *R* environment with Prolog widens the range of applications for logic programming and inductive logic programming. It has the potential to facilitate the development of systems combining logic and probabilistic reasoning and will significantly improve the development of ILP applications requiring statistical and numerical computations. We also hope that this interface will encourage logic programming researchers to engage in areas of research where a synergy of knowledge representation and statistical prowess is needed such as in bioinformatics and computational biology. Symmetrically, our library increases the tools available to *R* researchers and programmers who wish to exploit Prolog's advanced AI capabilities.

Possible extensions to the library include tighter integration with backtracking, although this has not been a limitation to the current applications. One specific aspect of such closer integration that might be of immediate value is the re-use of *hidden* variables (Section 3.3). An even tighter integration might be possible by allowing hidden and other *R* variables to be available for garbage collection. Finally, it would be interesting to investigate an even tighter syntactic integration by means of extensions to the syntax admitted by Prolog.

R..eal was originally designed, developed and tested on *YAP 6.3.1* under the Linux operating system. It has also been compiled for, and known to be working on MS operating systems and Mac OS. It was later ported [19] to the *SWI* [21] engine via a complete re-write of the *C* code. This has become the main development code as *YAP* provides a comprehensive compatibility layer to *SWI*'s *C* interface [20]. The library and examples presented here can be downloaded from our website (`http://bioinformatics.nki.nl/~nicos/sware/r..eal/`).

Acknowledgments. This work was co-financed by the Netherlands Consortium for Systems Biology (NCSB) which is part of the Netherlands Genomics Initiative / Netherlands Organisation for Scientific Research. VSC is funded by the ERDF through the Progr. COMPETE and by the Portuguese Gov. through FCT-Found. for Science and Tech., proj. LEAP ref. PTDC/EIA-CCO/112158/2009 and FCOMP-01-0124-FEDER-015008, and proj. ADE ref. PTDC/EIA-EIA/121686/2010 and FCOMP-01-0124-FEDER-020575.

References

1. Alves, A., Camacho, R., Oliveira, E.: Discovery of functional relationships in multi-relational data using inductive logic programming. In: IEEE Int. Conf. on Data Mining, pp. 319–322. IEEE Comp. Society, CA (2004)
2. Angelopoulos, N., Cussens, J.: Bayesian learning of Bayesian networks with informative priors. Journal of Annals of Mathematics and Artificial Intelligence 54(1-3), 53–98 (2008)
3. Angelopoulos, N., Shannon, P., Wessels, L.: Search and rescue: logic and visualisation of biochemical networks. In: Proceedings of the ICLP 2012 Workshop on Constraints in Bioinformatics (WCB 2012), Budapest, Hungary, pp. 1–6 (September 2012)
4. Angelopoulos, N., Taylor, P.: An extensible web interface for databases and its application to storing biochemical data. In: WLPE 2010, Scotland (July 2010)
5. Becker, R.A., Chambers, J.M., Wilks, A.R.: The New S Language: A Programming Environment for Data Analysis and Graphics. Wadsworth & Brooks/Cole, USA (1988)

6. Costa, V.S., Page, D., Qazi, M., Cussens, J.: CLP(BN): Constraint logic programming for probabilistic knowledge. In: Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI 2003), pp. 517–524 (2003)
7. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Journal of Theory and Practice of Logic Programming 12, 5–34 (2012)
8. Csardi, G., Nepusz, T.: The igraph software package for complex network research. InterJournal, Complex Systems 1695 (2006)
9. Cussens, J.: Stochastic logic programs: Sampling, inference and applications. In: Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI 2000), San Francisco, CA, pp. 115–122 (2000)
10. Dimitriadou, E., Hornik, K., Leisch, F., Meyer, D., Weingessel, A.: e1071: Misc Functions of the Department of Statistics (e1071), TU Wien (2011)
11. Gentleman, R.C., Carey, V.J., Bates, D.M., et al.: Bioconductor: Open software development for computational biology and bioinformatics. Genome Biology 5, R80 (2004)
12. Kimmig, A., Demoen, B., Raedt, L.D., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming 11, 235–262 (2011)
13. Murtagh, F.: Multidimensional Clustering Algorithms, COMPSTAT Lectures, vol. 4. Physica-Verlag, Wuerzburg (1985)
14. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, Previous number = SIDL-WP-1999-0120 (November 1999), `http://ilpubs.stanford.edu:8090/422/`
15. R Development Core Team. R: A Language and Environment for Statistical Computing. R Found. for Stat. Comp., Vienna, Austria (2011), `http://www.R-project.org/`
16. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. Journal of AI Research 15, 391–454 (2001)
17. Shannon, P.: RCytoscape: Display and manipulate graphs in Cytoscape. R package (2011)
18. Srinivasan, A.: The Aleph Manual. University of Oxford (2004)
19. Wielemaker, J., Angelopoulos, N.: Syntactic integration of external languages in Prolog. In: ICLP Workshop on Logic-based methods in Programming Environments (WLPE 2012), Budapest, Hungary, pp. 40–50 (September 2012)
20. Wielemaker, J., Costa, V.S.: On the portability of Prolog applications. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 69–83. Springer, Heidelberg (2011)
21. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming 12(1-2), 67–96 (2012)
22. Zhou, N.-F.: The language features and architecture of B-Prolog. Theory and Practice of Logic Programming 12, 189–218 (2012)

Reversible Language Extensions and Their Application in Debugging

Zoé Drey[1], José F. Morales[1], Manuel V. Hermenegildo[1,2], and Manuel Carro[1,2]

[1] IMDEA Software Institute, Madrid, Spain
[2] School of Computer Science, T. U. Madrid (UPM), Spain

Abstract. A range of methodologies and techniques are available to guide the design and implementation of language extensions and domain-specific languages on top of a base language. A simple yet powerful technique to this end is to formulate the extension via source-to-source transformation rules that are interleaved across the different compilation passes of the base language. Despite being a very successful approach, it has the main drawback that the input source code is lost in the process. As a result, during the whole workflow of program development (warning and error reporting, source-level debugging, or even program analysis) the tools involved report in terms of the base language, which is confusing to users. In this paper, we propose an augmented approach to language extensions for Prolog, where symbolic annotations are included in the target program. These annotations allow the selective reversal of the translated code. We illustrate the approach by showing that coupling it with minimal extensions to a generic Prolog debugger allows us to provide users with a familiar, source-level view during the debugging of programs which use a variety of language extensions, such as functional notation, DCGs, or CLP{Q,R}.

Keywords: language extensions, debuggers, logic programming, constraint programming.

1 Introduction

One of the key decisions when specifying a problem or writing a program to solve it is choosing the right language. Even when using recent high-level and multi-paradigm languages, the programmer often still needs precise, domain-specific vocabulary, notations, and abstractions which are usually not readily available. These needs are the main motivation behind the development of domain-specific languages, which enable domain experts to express their solutions in terms of the most appropriate constructs.

However, designing a new language can be an intimidating task. A range of methodologies and tools have been developed over the years in order to simplify this process, from compiler-compilers to visual environments [13]. A simple, yet powerful technique for the implementation of domain-specific languages is based on source-to-source transformations. Although in this process the source and target language can be completely different, it is frequent to be just interested in some *idiomatic extensions*, *i.e.,* adding domain-specific features to a host language while preserving the availability of most of the facilities of this language. Examples of such extensions are adding functional notation to a language that does not support it, adding a special notation for grammars (such as Definite Clause Grammars (DCGs) [16]), *etc.* Such transformations have been proposed in the context of object-oriented programming (*e.g.,* Polyglot for Java, [15]), functional programming (*e.g.,* Embedded DSL facilities for Haskell, [9]), or logic programming (*e.g.,* the term_expansion facility in most Prolog systems, or the extended mechanisms of Ciao [2,8]). In this approach, the language implementations provide a collection of *hooks* that allow the programmer to extend the compiler and implement both syntactic and semantic variations.

An important practical aspect is that, in addition to appropriate notation, the programmer also needs environments that help during program development. In particular, basic tools such as editors, analyzers, and, specially, debuggers are fundamental to productivity. However, in contrast to the significant attention given to mechanisms and tools for defining language extensions, comparatively few approaches have been proposed for the efficient construction of such development environments for domain-specific languages. In some cases ad-hoc editors, debuggers, analyzers, *etc.* have been developed from scratch. However, this approach is time consuming, error prone, hard to maintain, and usually not scalable to a variety of language extensions.

A more attractive alternative, at least conceptually, is to reuse the tools available for the target language, such as its debuggers or analyzers. This can in principle save much implementation effort, in the same way in which the source-to-source approach leverages the implementation of the target language to support the domain-specific extensions. However, the downside of this approach is that these tools will obviously communicate with the programmer in terms of the target language. Since a good part of the syntactic structure of the input source code is lost in the transformation process, these messages and debugger steps in terms of the target language are often not easy to relate with the source level and then the target language tools are not really useful for their intended purposes. For example, a debugging trace may display auxiliary calls, temporary variables, and obscure data encodings, with no trivial relation with the control or data domain at the source level. Much of that information is not only hard to read, but in most cases it should be invisible to the programmer or domain expert, who should not be forced to understand how the language at the source level is embedded in the supporting language.

In this paper, we propose an approach for recovering *symbolically* the source of particular translations (that is, *reversing* them and providing an *unexpanded* view when required) in order to make target language level development tools useful in the presence of language extensions. Our solution is presented in the context of Ciao [8], which uses a powerful language extension mechanism for supporting several paradigms and (sub-)languages. We augment this extension mechanism with support for symbolic annotations that enable the recovery of the source code information at the target level. As an example application, we use these annotations to parameterize the Ciao interactive debugger, so that it displays domain-specific information, instead of plain Prolog goals. Our approach requires only very small modifications in the debugger and the compiler, which can still handle other language extensions in the usual way.

The paper is organized as follows: Section 2 presents a concrete extension mechanism and illustrates the limitations of the traditional translation approach in our context. Section 3 presents our approach to unexpansion, and guidelines for instrumenting language extensions so that the intervening translations can be reversed as needed into their input source code. Section 4 presents the application of the approach to the case of debuggers. Finally, Section 5 presents related work and Section 6 concludes and suggests some future work.

2 Language Extensions and Their Limitations

We present a concrete language extension mechanism based on translations (the one implemented in the Ciao language) and then illustrate the limitations of the traditional translation-based extension approach in our context. In Ciao [8], language extensions are implemented through *packages* [2], which encapsulate syntactic extensions for the input language, translation rules for code generation to support new semantics, and the necessary run-time code. Packages are separated into compile-time and run-time parts. The compile-time parts (termed *compilation modules*) are only invoked during compilation, and are not included in executables, since they are not necessary during execution. On the other hand, the run-time parts are only required for execution and are consequently included in executables. This phase distinction has a number of practical advantages, including obviously the reduction of executable sizes.

More formally, let us assume that an extension for some language denoted as $\mathcal{L}_e$ is defined by package $PkgMod_e$, and that the compiler passes include calls to a generic expansion mechanism $[\![expand]\!]$, which takes a package, an input program in the source language, and generates a program in the target language $\mathcal{L}$. That is, given $[\![expand]\!]_e = [\![expand]\!](PkgMod_e)$, for a program $P_e \in \mathcal{L}_e$ we can obtain the expanded version $[\![expand]\!]_e(P_e) = P \in \mathcal{L}$. Note that in practice, Ciao contains finely grained translation hooks, which allow a better integration with the module system and the composition of translations [14]. This level of detail is not necessary for the scope of this paper; thus, for the sake of simplicity, the expansion will work on whole programs at a time.

Functional Notation. We illustrate the translation process in Ciao with an example from the *functional notation* package [3]. This package extends the language with *functional*-like syntax for relations. Informally, this extension allows including terms with predicate symbols as part of data terms, while interpreting them as predicate calls *with an implicit last argument*. It also allows defining clauses in functional style where the last argument is separated by a `:=` symbol (other functionalities are provided, such as expanding goals in the last argument after the body). The translation can be abstractly specified as a collection of rewrite rules such as:

$$\begin{array}{ll} \text{(Clauses)} & \textbf{tr}[\![\ p(\bar{a})\ \texttt{:=}\ C\ \texttt{:-}\ B\ ]\!] = (p'(\bar{v}, T)\ \texttt{:-}\ \bar{v}\ \texttt{=}\ \bar{a}, B, T\ \texttt{=}\ C) \\ \text{(Calls)} & \textbf{tr}[\![\ q(\ldots p(\bar{a})\ldots)\ ]\!] = (p'(\bar{a}, T), q(\ldots T \ldots)) \end{array}$$

The first rule describes the meaning of a clause in functional notation, where p' is the predicate in plain syntax corresponding to the definition of p in functional notation (*i.e.,* using `:=`). The second rule must be applied using a leftmost-innermost strategy for every p function symbol that appears in the goal q, where T is a new variable (skipping higher-order terms).

The usual evaluation order in logic programming corresponds to eager, call-by-value evaluation (but lazy evaluation is possible as shown in [3]). We refer to the actual implementation later in this section.

Example 1. The program excerpt below defines a predicate `f/2` in functional notation and its translation into plain Prolog code. Its body contains nested calls to `k/2` and `l/2`, and also syntactic sugar for a conditional (if-then-else) construct (using the syntax: *CondGoal* ? *ThenExpr* | *ElseExpr*) .

Source code (functional notation)	*Target code (plain Prolog)*
`f(X) := X < 42 ?`	`f(X,Res) :- X < 42, !,`
`        (k(l(m(X))) * 3)`	`   m(X, M), l(M, L), k(L, K),`
`      | 1000.`	`   T is K * 3, T = Res.`
`k(X) := X + 1.`	`f(X,1000).`
`l(X) := X - 2.`	`k(X,Res) :- Res is X+1.`
`m(X) := X.`	`l(X,Res) :- Res is X-2.`
	`m(X, X).`

Forgetful Translations and Loss of Symbolic Information. Both the standard compilation and the translations for language extensions are typically focused on implementing some precise semantics during execution. That is, the correctness of the translation guarantees that for all programs $P_e \in \mathcal{L}_e$, the expected semantics $[\![exec]\!]_e$ for that language can be described in terms of a program $P \in \mathcal{L}$ and its corresponding execution mechanism $[\![exec]\!]$. That is, for all $P_e \in \mathcal{L}_e$ there exists a $P = [\![expand]\!]_e(P_e)$ so that $[\![exec]\!]_e(P_e) = [\![exec]\!](P)$.

Most of the time, symbolic information at the source level is lost, since it is not necessary at run time. In particular, such information removal and loss of structure is necessary to perform important program optimizations (*e.g.,* assigning some variables to registers without needing to keep the symbolic name, its relation to other variables in the same scope, *etc.*). When programs are not executed, but manipulated at a symbolic level, the translation-based approach is no longer valid on its own. For example, assume a simple *debugger* that interprets the source and allows the user to inspect variable values at each program point interactively. In this case the translation, as a program transformation, must

$$\begin{tikzcd}
 & \boxed{PkgMod_e} \arrow[d] & & \boxed{DbgMod} \arrow[d] & \\
\boxed{P_e} \arrow[r] & [\![expand]\!] \arrow[r] & \boxed{P} \arrow[r] & [\![expand]\!] \arrow[r] & \boxed{P_{dbg}}
\end{tikzcd}$$

Fig. 1. The translation process and application of the standard debugger

```
2 2 Call: f(3,_6378) ?
3 3 Call: <(3,42) ?
4 3 Call: m(3,_6658) ?
5 3 Call: l(3,_6663) ?
6 4 Call: is(_6663,3-2) ?
...
9 3 Call: is(_6673,2*3) ?
10 3 Call: =(_6378,6) ?
```

Fig. 2. Excerpt of the display of the interactive debugger

preserve not only the input/output behaviour but also some other *observable* features (such as line numbers or variable names).

In order to explore the particular case of debuggers more closely, Figure 1 illustrates the translation process of a source program, using a compilation module $PkgMod_e$ containing the translation rules for extension e. If the developer asks the Ciao interpreter to debug this program, further instrumentation is applied that is also defined in part as a language extension, *DbgMod* in Figure 1; this instrumentation customizes the code by encapsulating it into a predicate that specifies whether a part of the code is *spy-able* or not. The following example illustrates in a concrete case the limitations of this process.

Example 2 (Interactive debugging). Consider the code and transformation of Example 1. If the target-level debugger is used without any other provision, following the process of Figure 1, debugging a call to `f(3,T)` amounts to debugging its translation, as illustrated in the trace of Figure 2 (the exit calls are omitted in order to save space). The problem of this trace is twofold: first, the interactive debugging does not make explicit the actual source-level predicate that is currently being tested. Second, understanding the trace forces the developer to make the mental effort of analyzing the debugged data and mapping it back to the source code. This effort increases if the source code contains operators that do not exist on the target (Prolog) side. The first case can be easily overcome when operator definitions are shared, *e.g.,* using a graphical editor and catching the operator with the line number and the occurence number of the call. However, the second case implies remembering the mapping between the source and the target operator. Furthermore, things get even more tedious when one instruction in the source language is translated into a composition of goals.

$$\begin{tikzcd}
\boxed{P_e} \arrow[r, "[\![expand]\!]_e"'] \arrow[d, "\mathsf{Obs}_e(i)"'] & \boxed{P} \arrow[d, "\mathsf{Obs}(i)"] \\
\boxed{V_e} & \boxed{V}
\end{tikzcd}
\qquad
\begin{tikzcd}
\boxed{P_e} \arrow[r, "[\![expand]\!]_e^{sym}"'] \arrow[d, "\mathsf{Obs}_e(i)"'] & \boxed{(P, Sym)} \arrow[d, "\mathsf{Obs}^{sym}(i)"] \arrow[dl, dashed, "\mathsf{Obs}_e^{sym}(i)"] \\
\boxed{V_e} & \boxed{V} \arrow[l, "unexp_v"']
\end{tikzcd}$$

Fig. 3. Observation problem at the source level (left); Observation using symbolic information (right)

3 Building Reversible Extensions

In this section we provide an informal definition of *unexpansion* with respect to a language extension. We then present guidelines in order to instrument a compilation module for such a language extension. This instrumentation aims to drive the process of reconstructing a program in terms of the language extension (or *source* language) in which the program is written. Through this mechanism, a language extension can be made *reversible.* To illustrate our purpose, we apply the guidelines and parameterize one of the translation rules used in the functional notation extension.

3.1 Expansion, Unexpansion, and Observers

We use the term *unexpansion* to designate the inverse of the expansion $[\![expand]\!]_e$, that is, the recovering of the original P_e source program from P. Unfortunately, this inverse is rarely a one-to-one mapping. For example, `f(3,T)` in $\mathcal{L}$ corresponds to both `T=f(3)` and `f(3,T)` (with `f/1` using functional notation). For another example, a clause can either be translated into one or many clauses, as depicted in Example 1 for `f` in functional notation.

Having multiple solutions for unexpansion can be confusing for the user and impractical for automatic transformations. However, the most important use of unexpansion in our context is to observe the behavior of only certain program aspects at the source language level. In this case, unexpansion seems more treatable. For that purpose we define the term *observer* accordingly: an *observer* is an interface that provides some specific source-level information about a particular program. The observer can be either static or dynamic. Specifically, we can consider as observers monitors (*e.g.,* interactive debuggers, tracers, and profilers) for dynamic observation, and verifiers (*e.g.,* static analyzers and model checkers) for static observation. Thus, a source-level view may correspond to the current instruction being invoked in an interactive debugger, or to a trace of the memory state, in a tracer, or perhaps the dependencies between the program variables, in a static analyzer, all of them represented in terms of the source language abstractions.

The correspondance between expansion and unexpansion, in the context of an observer, is sketched in Figure 3. We assume that we have observers $\mathsf{Obs}_e(i)$ and

$\mathsf{Obs}(i)$ for the source and target languages, respectively. We denote by i some particular observable aspect and by V the aspect (*e.g.,* "line numbers" and an integer). On the left diagram we depict the impossibility of getting information at the $\mathcal{L}_e$ level in general. To provide the programmer with source-level observers, our approach relies on extending the expansion ($[\![expand]\!]_e^{sym}$) with additional symbolic information (which can be significantly smaller than the sources). Then, observers $\mathsf{Obs}^{sym}(i)$ can retrieve V (*e.g.,* a single number encoding the row and columns) and map it back to V_e (*e.g.,* the row and columns). This composition provides an effective $\mathsf{Obs}_e^{sym}(i)$.

We now propose guidelines for easily instrumenting the translation module of a language extension, in such a way that observers can be parameterized with respect to this instrumentation.

3.2 Instrumentation of a Compilation Module

Instrumenting a compilation module involves annotating its translation rules with source code information that can then be used by an observer (*i.e.,* the debugger in our application example). We illustrate the instrumentation process on the functional extension example.

Guidelines. The first step in making a language extension reversible is to determine which parts of the source code need to be kept available in the expansion process. The second step is to determine how and where to propagate this information, so that it can be accessed whenever the developer requires observation during program execution. The third step is to determine the representation of the observable data.

Event and data analysis. What events do we want to observe? What do we want to observe about them? These selections should be useful for following the control flow and state changes during program execution. For example, in a λ-calculus-like language, the definition and the application of a function are two of the key elements to follow in order to debug a program [17]. As another example, in a goal involving expressions in functional notation, the debugger must be aware of which positions correspond to data terms and which positions to predicate calls.

Decomposition. How is a source statement decomposed into target code? The answer to this question implies in part how the data that we want to observe should be propagated. For example, while the generic debugger may step through a number of target-level statements, a source-specific debugger may have to consider a single source statement as corresponding to all those steps. This applies for example in the conditional statement `C ? A | B` of the functional notation, where `A` is translated into an (at least) two-goal target code segment.

Representation. How should the data to be observed be represented? In a purely syntactic extension, data always represents elements of the concrete syntax. Nevertheless, it is interesting to consider this question when displaying the runtime

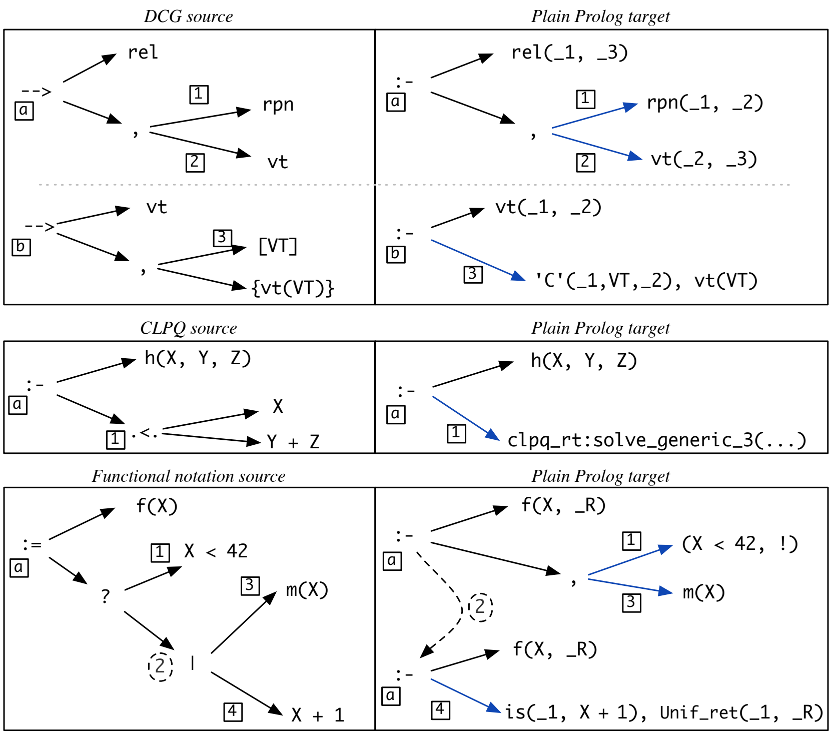


Fig. 4. Annotated translation: source-level code (left) and target-level code (right)

context, such as the state of the memory, for semantic extensions. For example, in a CLP{Q,R} extension, variables are bound at run-time to complex terms attached to attributed variables which reflect the internal, low-level representation of the constraint store, while what the programmer would like to see is a symbolic representation of the constraints among the variables in the source constraint language.

Some Examples. We now present three excerpts of code written using language extensions, namely CLPQ (the case for CLPR is analogous), DCG, and the functional notation. The source-level code is represented as a concrete syntax tree in Figure 4, left part. It is translated into Prolog code as depicted in Figure 4, right part.

The annotation process implies tagging each element of the source code that is intended to be observed, *i.e.,* each element meant to be or to refer to a "first-class" concept of the language extension, whether it is defined as a clause or a goal. For example, in a DCG rule such as "`rel --> rpn, vt`" (Figure 4, top part), `rpn` and `vt` are annotated, as these functors lead to a call to another grammar rule. In contrast, elements surrounded by brackets (*e.g.,* `{vt(VT)}`) correspond to plain Prolog, and thus do not require any source level-related special handling. During the expansion process of the source-level code, the target goals correspondingly keep a reference to the source identifiers.

3.3 Implementation of the Instrumentation Process

To instrument the translation rules we propose to annotate the target parameter of each rule (*i.e.,* the argument in which the code generated by the translation is returned). This annotation is defined as a predicate and provides the symbolic information, encoded as a Prolog term, to drive the process of recovering source code data within the observer. Symbolic information could be a list of variables (along with a function that recovers their value at the source level from the target context, *i.e.,* its environment and store), or a single string to be displayed at the observer output at run time, *e.g.,* in the case of an interactive debugger.

Definitions. We currently distinguish two types of annotations: the `$clause_info` annotation, which is wrapped around target clauses, and the `$goal_info` annotation (or meta-information), which is wrapped around target goals. The purpose of each of these annotations is to gather symbolic information to recover a source-level statement or a source-level call, respectively. Additionally, this distinction enables handling clauses and goals properly, in particular to retrieve their location in source modules.

Both annotations are handled according to the wrapped target element. Specifically, `$clause_info` takes three arguments: the target clauses (`Clauses` in Example 3), a source-level representation (`SI`), and an identifier (`Id`) to optionally enable later retrieval of the translated statement. The body of the source-level statement is itself tagged so as to map to the corresponding target goals when the statement is evaluated. The `$goal_info` annotation takes two arguments: the target goal or goal composition, and an identifier, enabling the retrieval of the source-level call in the body of a statement.

Application to the Functional Notation Translation. We illustrate this annotation process with Example 3. Specifically, we instrument the two translation predicates `defunc` and `defunc_goal`, translating the source-level clauses and goals. The text in italic corresponds to the instrumentation code added over the original translation predicates. Note that defining this code's body (according to the data to observe) can defined as a hook of an extension module.

Example 3. Instrumentation of the translation rules for functional notation.

```
defunc((FuncHead := FuncVal), $clause_info(Clauses, Id, SI) :−
    identify_functional_calls(FuncVal, FuncVal_withIds),
    defunc_rec((FuncHead := FuncVal_withIds), Clauses, SI0),
    build_syminfo([FuncHead,':='|SI0],SI).

defunc_rec((FuncHead := FuncValOpts_withIds), Clauses, [SI1|SIR]) :−
    FuncValOpts_withIds = (FuncVal1 | FuncValR), !,
    Clauses = [Clause1 | ClauseR],
    defunc_rec((FuncHead := FuncVal1), Clause1, SI1), (1)
    defunc_rec((FuncHead := FuncValR), ClauseR, SIR). (2)

defunc_goal(FuncCall, Goal) :−
    recover_id(FuncCall, Id),
    normalize(FuncCall, NormGoal),...,
    Goal = 'Eval'('$goal_info'(NormGoal, Id), _Ret_Arg).
```

Clause translation. The `FuncHead` part on the left corresponds to a predicate declaration; the `FuncValOpts` part on the right corresponds to goal invocations (this results from the data analysis guideline). As introduced in Section 3.2, the source-level elements of the predicate body are tagged using the user-defined functor `identify_functional_calls`. Specifically, each relevant element of the body is associated with a unique identifier, in order to enable the retrieval of its position by the observer.

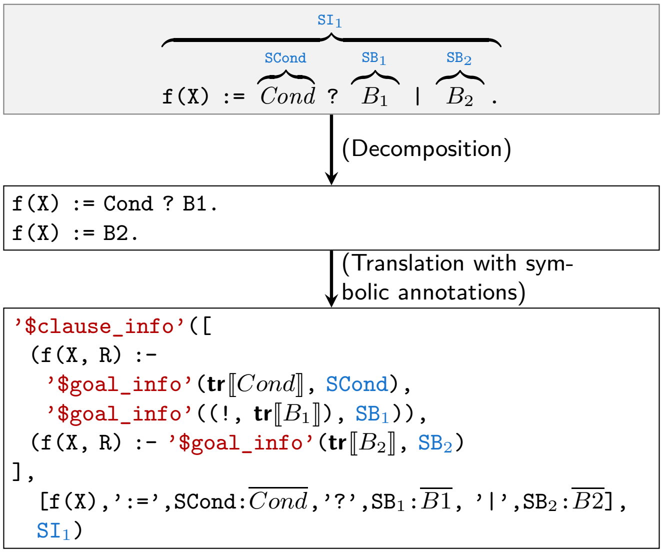
The symbolic information attached to the annotation is represented by the contents of variable `SI`, created by predicate `build_syminfo`. This variable is handled by an observer, according to the nature of the program view it aims to provide. For example, line numbers, variables, or function names can be attached to it. It can even be left as a free variable, if the observer can automatically retrieve the information.

Notice that the declaration `FuncHead := FuncValOpts` is decomposed into many goals, marked *(1)* and *(2)*, if the `|` operator appears inside its right part. Therefore, the translation needs to indicate to the observers that the declaration is to be treated as a single one. This is done by grouping the symbolic information computed by the evaluation of goals *(1)* and *(2)* into the `$clause_info` wrapper, set as its last argument in the first predicate `defunc`.

Goal translation. To relate target goals with their symbolic information the goal translation predicates are instrumented with (1) predicate `recover_id` which extracts identifier `Id` of the annotated source-level goal, and (2) predicate `$goal_info` which is wrapped around the translation code `NormGoal`. `Eval` is not wrapped since it is an intermediate goal in the expansion process.

This approach based on symbolic information enables us to envision a range of program views, from simple syntax recovery to high-level representation of analysis results: annotations can be enriched with source-specific procedures to handle various representations of the target program, enabling different instantiations of the annotation variable. They can even hold procedures that perform advanced computations parameterized with the symbolic information (*e.g.,* counting the number of times a function is invoked).

The instrumentation method is outlined in the schema of Figure 5, which depicts a declaration of the form `f(X) :=` *Cond* `?` B_1 `|` B_2. In this figure, the variable names Sx correspond to identifiers of some program elements associated with some symbolic information, and the expressions **tr**$[\![x]\!]$ correspond to a translation of the term x. Overlined elements represent syntactic nodes.



```
f(X) := Cond ? B1 | B2 .
```

```
f(X) := Cond ? B1.
f(X) := B2.
```

```
'$clause_info'([
  (f(X, R) :-
    '$goal_info'(tr⟦Cond⟧, SCond),
    '$goal_info'((!, tr⟦B1⟧), SB1)),
  (f(X, R) :- '$goal_info'(tr⟦B2⟧, SB2)
],
   [f(X),':=',SCond:Cond,'?',SB1:B1, '|',SB2:B2],
   SI1)
```

Fig. 5. Instrumented translation of a clause in functional notation

Implementation Overview. The annotations are integrated into the compilation process of an extension module. The overall process of making program behavior observable at the source level through an observer is depicted in Figure 6. Let us describe the extractor and the controller parts of this process.

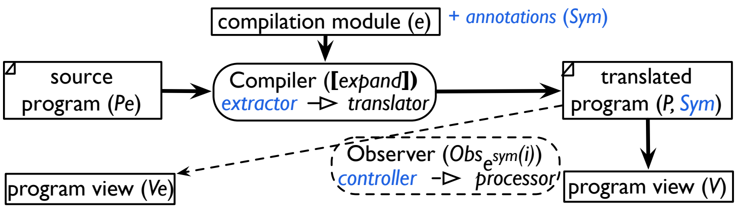


Fig. 6. Overview of the compilation process extended with annotations

Extending the translation process: extractor. Handling the annotations required a slight extension of the translation phase of the compiler. The extension is as follows: the `$clause_info` annotation is handled inside the original processing step of clauses; the `$goal_info` annotation is handled in the processing step of goals. Specifically, the annotation attached to each clause is encapsulated into an auxiliary clause generated in the same phase as the regular target clause(s). In doing so, the symbolic information is available whenever an observer needs it. During the goal translation step, the target goals that are wrapped by a `$goal_info` annotation are kept executable, enabling the annotation to be ignored whenever the source code needs to be processed by a target-level observer.

Customizing the observer: controller. Existing observers for Prolog (such as debuggers, profilers, or static analyzers) can then be customized according to the generated symbolic information. This customizing process is necessarily application-specific, but we provide some general hints as guidelines:

(Hint 1) Collect all the elements requested by the observer (*e.g.,* line numbers, function names, or some text) and make them accessible as a data structure in the arguments of the observer predicates.

(Hint 2) Complement the observer functions that do some processing on some element (clause or goal), by checking if this element holds some symbolic information (*i.e.,* as `$clause_info` or `$goal_info`). For example, if a `$goal_info` wrapper is encountered, extract the name of the source-level goal and make it display it instead of the name of its corresponding target-level goal(s). Otherwise, preserve the target-level behavior.

(Hint 3) Identify locations in the observer where it needs to remember the last data computed with source-level information. For example, such data can correspond to a counter of execution times of a given function, when the observer is a profiler.

4 Application to the Interactive Debugger

We now illustrate the use of a reversible language extension to parameterize the generic interactive debugger of Ciao. We describe the modifications performed on the debugger, and show the resulting source-level trace for our initial example (Example 1).

4.1 Implementation Details

In the case of the debugger, the required symbolic information corresponds to a source node (*e.g.,* `[k(X), ':=', +(X, 1)]` as in Example 1). As a result, the extraction process consists solely of storing each source node before its expansion.

Once the source-level information is extracted and mapped to the appropriate target term (or composition of target terms, *cf.* the guidelines in Section 3), it is interpreted by the debugger. To step through the source code instead of the target code, the `controller` part of the debugger checks for the presence of a meta-information call at the level of the translated program (Hint 1), and displays a trace step accordingly. In particular, it is responsible for locating the name and execution counter of the target goal in the nodes corresponding to this goal, and for replacing it with the related symbolic information, *e.g.,* the name and the counter of the source-level goal associated with this target goal (Hint 2). Note that if a source-level goal maps to a composition of goals, the controller will behave as if only one step occurs, hiding the underlying target goals in the trace display either until another annotated goal is encountered, or until the last target-level goal has been (silently) executed. The information necessary to this source-level step is stored, in order to refer to it in a later step (*e.g.,* exit or failure

```
2 2 Call: ex0:f(3,_6371) ?
3 3 Call: f(3) := 3 < 42  ? k(l(m(3))) * 3 | 1000 ?
4 4 Call: f(3) := 3 < 42 ? k(l(m(3))) * 3 | 1000 ?
5 5 Call: m(3) := 3 ?
6 4 Call: f(3) := 3 < 42 ? k(l(m(3))) * 3 | 1000 ?
7 5 Call: l(3) := 3 - 2
8 4 Call: f(3) := 3 < 42 ? k(l(m(3))) * 3 | 1000 ?
9 5 Call: k(1) := 1 + 1
10 3 Call: f(3):= 3 < 42 ? k(l(m(3))) * 3  1000 ?
2 2 Exit: ex0:f(3,12) ?
```

Fig. 7. An excerpt of the debugger trace, customized with source information

step implying backtracking) (Hint 3). When a goal invoked in the debugger is neither annotated nor part of an annotation, the controller executes its original procedure to display the standard, expanded debug information.

4.2 Source-Level Tracing: The Functional Example Revisited

With this instrumentation, Example 1 is now debugged in source code terms, as illustrated in Figure 7. Note that the debugger now displays the complete declaration (see second line) defining `f`, instead of a single part of a clause (see the second line in Example 1). When a function evaluation returns a value (which is the case of all the functions `f/1`, `k/1`, `l/1`, `m/1`), intermediate unifications are performed by the generic debugger. When the debugger is instrumented with a controller (*i.e.,* the handler of annotations), these unification steps are ignored (skipped over), since they have no representation in the original source code.

5 Related Work

There exist frameworks and generative approaches that facilitate the development of DSL tools for programming, including debuggers [6,20]. For example, the Eclipse Integrated Development Environment [6], provides an API and an underlying framework that can greatly help in the development of a debugger [5]. Emacs is another example of such environments, with facilities in the same line as Eclipse. However, these tools are large and have a significant learning curve, and, more importantly, their facilities are centered more around the graphical navigation of the source code and interfacing with a command-line debugger, while the focus of our work is on bridging syntactic or semantic aspects between two sides of a translation, within such a command-line debugger. In that sense our work is complementary to (and in practice combines well with) the facilities in Eclipse, Emacs, and related environments. Generative approaches have been suggested (*e.g.,* based on aspect weaving into the language grammar [22]) in order to reduce developer burden when using intricate APIs.

However, none of these approaches provide a methodology for developing reliable and maintainable debuggers. As a result, the development of debuggers has remained difficult, inciting DSL tool developers to implement ad-hoc solutions, through extension-specific modifications and adaptations of the debugger code. For example, SWI-Prolog and Logtalk include a debugger for Prolog with built-in support for language extensions like DCGs programs [21], which is purely based on storing line numbers within the code. As mentioned in the introduction, this approach, although useful in practice, is limited to a reduced kind of extensions.

Our objective has been to develop a more general approach, which we have illustrated by applying the same methodology to several extensions including functional notation, DCGs, and CLP{Q,R}.

Lindeman *et al.* [11] have proposed recently a declarative approach to defining debuggers. To this end, they use SDF [19], a rewriting system, to instrument the abstract syntax tree with debugging annotations. However, it does not seem obvious that their approach could be applied to other observer tools. Indeed, instrumentation is achieved by providing debugger-specific information, in the form of events. In contrast, our instrumentation process makes it possible to easily add and handle different kinds of meta-information.

Unexpansion and decompilation only differ in the hypothesis used in decompilation: that the original source code may not be available. It is interesting however to compare to existing related decompilation approaches. Bowen [1] proposes a compilation process from Prolog to object code which makes it possible to define decompilation as an inverse call to compilation, provided some reordering of calls is performed. Gomez *et al.* [7] also propose a decompilation process for Java based on partial evaluation. However, these approaches have not been designed to be applicable to a large class of different language extensions. More generally, while it is in theory possible (although predictably hard with current technology) to implement fully reversible transformations, this approach runs into the problem that such inversions are non-deterministic in general, in the sense that a given target code can be generated from multiple source texts. Presenting the programmer with a different code that what is in the source program could be even more confusing that debugging the target code directly.

More similar to our solution is the approach of Tratt [18], which also targets language extensions, and where source information is injected into the abstract syntax tree of the source program. This information is exploited to report errors in terms of the language extension. However, they only discuss how to inject such information in the syntax tree, and do not explain how to use this information when building or adapting tools.

The macro-expansion passing style [4] approach makes it possible to easily implement observers. Our approach differs from this one by relying on the existing generic debugger (Ciao's in our examples): it focuses on what changes are required in the debugger and the extension framework so that symbolic information for unexpansion is handled in a way that is independent from the concrete language extension.

6 Conclusion and Future Work

We have presented a generic approach that enables a debugger for a target language to display trace information in terms of the language extension in which a source program is written, using the Ciao debugger as an example. The proposed approach is based on an extension of the usual mechanisms for term expansion, and in particular of their modular implementation in Ciao through *packages*. Nevertheless, we believe that our proposal could be ported to other Prolog systems with minor modifications. More specifically, we have defined a methodology for making relevant parts of the source text and other characteristics available at the target level by enriching the translation rules. We have shown that the compiler and the debugger require only small adaptations in order to take this mechanism into account; these adaptations are generic in the sense that while the transformation rules are specific to the extension, the compiler and the debugger do not require further modification, for what is arguably a large class of extensions. In particular, in the paper we have illustrated this approach by applying it on the functional notation. In the system, we have successfully applied it also to the DCG and CLP{Q,R} constraint packages.

In future work, we plan to extend the flexibility of the approach by enriching the annotations to serve different purposes, such as performing computations on symbolic information. Also, we feel that this initial work on augmenting the language extension mechanism already provides us with the basis for adapting the Ciao pre-processor. In doing so, errors, warnings, or other reports are reported in terms of the source, domain-specific language, for different extensions, without requiring further modification of the pre-processor itself. The same would apply of course to the auto-documenter and the profiler [12]. Finally, we believe we could leverage Kishon *et al.*'s framework [10] to check the soundness of our approach with regard to the intended semantics of a language extension. Doing so would also make it possible to show the equivalence between the behavior of an ad-hoc source level debugger and our customization of the target level debugger.

References

1. Bowen, J.P.: From programs to object code and back again using logic programming: Compilation and decompilation. Journal of Software Maintenance Research and Practice 5(4), 205–234 (1993)
2. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: Lloyd, J., et al. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 131–148. Springer, Heidelberg (2000)
3. Casas, A., Cabeza, D., Hermenegildo, M.V.: A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 146–162. Springer, Heidelberg (2006)
4. Dybvig, R.K., Friedman, D.P., Haynes, C.T.: Expansion-passing style: A general macro mechanism. Lisp and Symbolic Computation 1(1), 53–75 (1988)
5. Eclipse. How to write an Eclipse debugger, `http://www.eclipse.org/articles/Article-Debugger/how-to.html`

6. ECRC. Eclipse User's Guide. European Computer Research Center (1993)
7. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Decompilation of Java Bytecode to Prolog by Partial Evaluation. Information and Software Technology 51(10), 1409–1427 (2009)
8. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming 12(1-2), 219–252 (2012)
9. Hudak, P.: Building domain-specific embedded languages. ACM Comput. Surv. 28(4es), 196 (1996)
10. Kishon, A., Hudak, P., Consel, C.: Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In: Proceedings of the ACM Conference on Programming Language Design and Implementation, PLDI, pp. 338–352. ACM Press, Toronto (1991)
11. Lindeman, R.T., Kats, L.C., Visser, E.: Declaratively defining domain-specific language debuggers. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering, GPCE 2011, pp. 127–136. ACM, New York (2011)
12. Mera, E., Trigo, T., Lopez-García, P., Hermenegildo, M.: Profiling for run-time checking of computational properties and performance debugging in logic programs. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 38–53. Springer, Heidelberg (2011)
13. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)
14. Morales, J.F., Hermenegildo, M.V., Haemmerlé, R.: Modular Extensions for Modular (Logic) Languages. In: Vidal, G. (ed.) LOPSTR 2011. LNCS, vol. 7225, pp. 139–154. Springer, Heidelberg (2012)
15. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
16. Pereira, F., Warren, D.: Definite clause grammars for language analysis - A survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence 13, 231–278 (1980)
17. Tolmach, A.P., Appel, A.W.: A debugger for standard ML. J. Funct. Program. 5(2), 155–200 (1995)
18. Tratt, L.: Domain specific language implementation via compile-time metaprogramming. ACM Trans. Program. Lang. Syst. 30(6), 31:1–31:40 (2008)
19. den van Brand, M.G.J., et al.: The Asf+Sdf meta-environment: A component-based language development environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
20. van den Brand, M.G.J., Cornelissen, B., Olivier, P.A., Vinju, J.J.: Tide: A generic debugging framework — tool demonstration. Electron. Notes Theor. Comput. Sci. 141(4), 161–165 (2005)
21. Wielemaker, J.: SWI-prolog — source-level debugger, `http://www.swi-prolog.org/gtrace.html`
22. Wu, H., Gray, J., Roychoudhury, S., Mernik, M.: Weaving a debugging aspect into domain-specific language grammars. In: Proceedings of the 2005 ACM Symposium on Applied Computing, SAC 2005, pp. 1370–1374. ACM, New York (2005)

proSQLite: Prolog File Based Databases via an *SQLite* Interface

Sander Canisius[1], Nicos Angelopoulos[1,2], and Lodewyk Wessels[1,2]

[1] Bioinformatics and Statistics, Netherlands Cancer Institute, Amsterdam, Netherlands
[2] The Netherlands Consortium for Systems Biology (NCSB)
{s.canisius,n.angelopoulos}@nki.nl

Abstract. We present a succinct yet powerful interface library to the *SQLite* database system. The single file, server-less approach of *SQLite* along with the natural integration of relational data within Prolog, render the library a useful addition to the existing database libraries in modern open-source engines. We detail the architecture and predicates of the library and provide example deployment scenarios. A simple bioinformatics example is presented throughout to illustrate *proSQLitet*'s main functions. Finally, this paper discusses the strengths of the system and highlights possible extensions.

Keywords: databases, *SQL*, Prolog libraries, *SQLite*.

1 Introduction

SQLite [1] is a powerful, open source server-less database management system that requires no configuration as its databases are stored in a single file. Ran from a lightweight operating system (OS) library executable, it can be deployed in a number of scenarios where a traditional server-client database management system (DBMS) is not possible, advisable or necessary. This paper presents an implementation of a Prolog library that uses the C-interface to communicate with the *SQLite* OS library.

The relational nature of Prolog makes its co-habitation with relational database systems an attractive proposition. Not only databases can be viewed and used as external persistent storage devices that store large predicates that do not fit in memory, but it is also the case that Prolog is a natural choice when it comes to selecting an inference engine for database systems. The *ODBC* library in *SWI-Prolog* [18] is closely related to our work since we have used the library as a blue print both for the C-interface code and for the library's predicates naming and argument conventions.

The field of integrating relational databases has a long tradition going back to the early years of Prolog [8]. For instance the pioneering work of Draxler[7], although based on writing out *SQL* rather than directly interrogating the database, provided extensive support for translating combinations of arbitrary Prolog and table-associated predicates to optimised *SQL* queries. The code has been ported to a number of Prolog systems[13]. Another approach which targeted machine learning and tabling as well as importing tables as predicates is *MYDDAS*, [5]. An early *ODBC* interface for *Quintus* Prolog was *ProDBI* [12]. Prolog has also been used to implement a database management system based on the functional data model [10]. In this contribution we concentrate on describing an open-source modern library that can be used out-of-the-box with

Table 1. Predicates for *proSQLite* library. Left: connection management and SQL queries. Right: auxiliary predicates on formatted queries and database introspection.

predicate name/arity	moded arguments	predicate name/arity	moded arguments
sqlite_connect/2	+File, ?Conn		
sqlite_connect/3	+File, ?Conn, +Opts	*sqlite_format_query/3*	+Conn, +SQL, -Row
sqlite_disconnect/1	+Conn	*sqlite_current_table/2*	+Conn, -Row
sqlite_current_connection/1	-Conn	*sqlite_table_column/3*	+Conn, ?Table, -Column
sqlite_default_connection/1	-Conn	*sqlite_table_count/3*	+Conn, +Table, -Count
sqlite_query/2	+SQL, -Row		
sqlite_query/3	+Conn, +SQL, -Row		

a zero configuration, community supported database system. We hope that the library will be a useful tool for the logic programming community and provide a solid basis in which researchers can contribute rather than having to reinvent the basic aspects of such integrations.

2 Library Specifics

Here we present the overall architecture of the system along with the specific details of the three component architecture. Our library was developed on *SWI 6.1.4* under a *Linux* operating system. It is also expected to be working on the *Yap 6.3.2* [6] by means of the *C*-interface emulation [16] that has been also used in the porting other low-level libraries [2]. We publish[1] the library as open source and we encourage the porting to other Prolog engines as well as contributions from the logic programming community to its further development. Deployment is extremely simple and only depend on the location of the *SQLite* binary.

Our library is composed of three main components. At the lower level, written in *C*, the part that handles opening, closing and communicating with the *SQLite* OS library. The *C* code is modelled after, and borrows crucial parts from the *ODBC* library of *SWI*. On top of the low-level interface, sit two layers that ease the communication with the database. On the one hand, a set of predicates allow the interrogation of the database dictionary, while a third layer associates tables to Prolog predicates.

The heart of the library is its interface to *SQLite*. This is implemented in *C* and has strong affinity to the *ODBC* layer in *SWI*. The left part of Table 1 lists the interface predicates to the core system. Management predicates allow users to open, close and interrogate existence of connections to databases. The *C* code creates a unique, opaque term to keep track of open connections. However, this is not particularly informative to the users/programmers. More conveniently, the library allows for aliases to connections that can act as mnemonic handles. As a running example we will use the connection to a large but simple protein database[2] from *Uniprot*. It has two tables referenced on a single key and having $286,525$ and $3,044,651$ entries. The single file *SQLite* database is $184Mb$ in size. Table 2 summarises the basic parameters of the database

[1] `http://bioinformatics.nki.nl/~nicos/sware`

[2] `http://bioinformatics..nki.nl/~nicos/sware/prosqlite/uniprot.sqlite`

Table 2. Structure of the uniport example database which stores protein identifier maps

table name	population	columns
secondary_accessions	286525	secondary_accession, primary_accession
identifier_mapping	3044651	uniprot_accession, identifier_type, target_identifier

The type of connection we wish to establish to the database file is controlled by *sqlite_connect/3* . The user can interrogate all open connections and the existence of a specific connection via *sqlite_current_connection/1.* This predicate backtracks over all open connection if it is queried with a variable as its argument. The bulk of the traffic with *SQLite* is directed via *sqlite_query/2,3* through which data in tables can be added, deleted and queried. We include in the library a small number of predicates that assist user interaction with databases. These include parametrised query strings, interrogating the database dictionary and simple aggregate operations.

The formatted query mechanism provides a means for parametrised queries. This is useful for encoding common patterns of queries in an application. The function of the rest of the wrapping predicates follows directly their naming. The information they provide is gathered from the database dictionary which is managed by *SQLite*. For illustration purposes and for comparison with alternative ways of obtaining the counts of a table, we show in the code that follows how to use backtracking to obtain all tables in the *Uniprot* database along with their populations.

```
?- sqlite_current_table(uniprot, Table),
   time(sqlite_table_count(uniprot, Table, Count)),
   write(Table:Count),nl,fail.

% 7 inferences,0.007 CPU in 0.007secs (99% CPU,1013 Lips)
secondary_accessions:286525
% 7 inferences,0.083 CPU in 0.083 secs (100% CPU,85 Lips)
identifier_mapping:3044651
```

2.1 Tables as Predicates

With the *as_predicates/1* option of *sqlite_connect/3* we can direct the library to create linking predicates for each table in the database. That is a predicate is created for each table in the underlying database. The predicates are created at module identified by option *at_module/1.* It is the responsibility of the user to ensure there are no name clashes. Once thus declared, the table predicates behave as normal Prolog predicates. The system makes simple transformations when filling the predicates with results from the database. Currently, this is by mean of creating an *SQL* `SELECT` statements in which the `WHERE` sub-clause is formed from the ground arguments of the corresponding goal. For a table with name `Name` and columns that have a one-to-one correspondence with the list of variables in `Args,` and `Module` being the module provided at the `at_module` option of *sqlite_connect/3* . Predicates that correspond to database tables interact as if defined by a number of facts: each table row corresponds to a fact assertion to the Prolog database. To illustrate, we show the predicated goals for the two queries from the preceding section. Times are shown from a run on a Linux dual-core $3.16 GHz$ desktop computer.

```
?- findall(S, secondary_accessions(S, 'P64943'), All).
   All = ['A0A111', 'Q10706'].

?- sqlite_current_table(uniprot, Table),
   findall(C,sqlite_table_column(uniprot,Table,C),Cs),
   length( Cs, Arity ), length( As, Arity ),
   Pred =.. [Table|As],
   time( ( findall(1,Pred,Ones),
               length(Ones,Count))),
   write(Table:Count), nl, fail.

%286,560 inf,0.561 CPU in 0.575 secs(98% CPU,510692Lips)
secondary_accessions:286525
%3,044,689inf,10.486CPUin10.516secs(100%CPU,290360Lips)
identifier_mapping:3044651
```

Predicated tables only depend on *SQL* transformations and as such are not specific to *SQLite* but can be easily ported to other interface libraries such as *ODBC*.

3 Applications

The last decades have witnessed a phenomenal increase in the amount of biological knowledge that has been published and codified [9]. This acceleration can be directly attributed to the evolution of high throughput technologies such as genome wide expression assays, microscopy and deep sequencing. One important way in which biological knowledge is codified is in the form of databases and ontologies. These include protein-protein interaction databases such as STRING [14] and HPRD [11] and protein information databases such as the universal protein resource *Uniprot* [15].

Prolog is a powerful platform for bioinformatics research and analysis. Its ability to query relational datasets and express recursive searches succinctly are of particular interest to ontologies and databases tabulating millions of relations. One of the main roadblocks hindering the use of Prolog in this research area is the lack of effective tools that give access to the resources available. Databases form the basic layer of biological knowledge available. The use of effective tools to connect databases in efficient, resilient and integrative manners to the logic engine can assist in narrowing this gap. Currently, we use *proSQLite* as one of the possible caching mechanisms in *pubGraph* a graph search tool that mines the citation relations from the *PubMed*[3] website to built visualisations of the relational networks.

Engaging Prolog with the world wide web (WWW) in the role of a web-server has been well advocated and served by supporting libraries [4,17]. Furthermore, there has been previous motivating work on systems that realise Prolog servers that mediate the

[3] `http://www.ncbi.nlm.nih.gov/pubmed`

web-publication of material stored in relational databases [3]. The library presented here further facilitates the role of Prolog in this area. Particularly, with small to medium size web services. The main benefits of *SQLite* in this context are:

persistence - Prolog based servers need persistent storage of data. It is conceivable that such data can be realised as external files managed privately.

threading - Web-servers are inherently multi-threaded and the ability to communicate through a shared file-based database provides further plurality.

ease of deployment - *SQLite* is arguably one of the easiest database back-ends to install and maintain.

filestore abstraction - One of the main success stories for *SQLite* has been in providing application specific filestore solutions. This fits well within a web-server setting.

Currently, a large number of applications are reported to be using *SQLite* as an embedded database transaction system that is used to store application data in a uniform and robust manner. These include major open source projects such as the *Firefox/Mozilla*[4] browser and the *Powerdns*[5] DNS server. The embedded nature of *SQLite* reduces overheads and simplifies installation. Applications can use the layer to abstract their interactions with the operating system. Databases are stored in single files and are cross-platform compatible.

4 Conclusions

We presented a stable and efficient library for integrating a file-based DBMS to modern open-source Prolog engines. We have argued that Prolog is a powerful platform for data analysis and computational research in bioinformatics and for the realisation of agile web-servers that require minimal programming effort. Biological knowledge captured in the growing list of databases can be efficiently reasoned with, within logic programming. There are a number of possible extensions that can be envisaged on top of the presented library. These are not necessarily specific to this library but can also be of relevance to similar approaches such as the *ODBC* library of *SWI*. One such extension is *d*b_facts[6] which implements term based table interactions for *proSQLite* and *ODBC* databases. It also allows for a notation that selects columns from tables independently of their position in the respective table. This would allow decoupling of a table's precise list of constituent columns from accessing specific fields, making code easier to maintain as additions to the database structure do not need to be propagated to parts of the code that are not accessing the new columns.

Acknowledgements. We would like to thank Jan Wielemaker for discussions and for making the *ODBC* library of the *SWI-Prolog* open source. Also we thank Graham Kemp for valuable comments. This work was (co)financed by the Netherlands Consortium for Systems Biology (NCSB) which is part of the Netherlands Genomics Initiative/Netherlands Organisation for Scientific Research. SC and NA contributed equally.

[4] `http://www.mozilla.org/`

[5] `http://doc.powerdns.com/gsqlite.html`

[6] `http://bioinformatics.nki.nl/~nicos/sware/db_facts`

References

1. Allen, G., Owens, M.: The Definitive Guide to SQLite (2010)
2. Angelopoulos, N., Costa, V.S., Camacho, R., Wielemaker, J., Azevedo, J., Wessels, L.: Integrative statistics for logical reasoning (2012), Conditional accept PADL 2013
3. Angelopoulos, N., Taylor, P.: An extensible web interface for databases and its application to storing biochemical data. In: WLPE 2010, Edinburgh, Scotland (July 2010)
4. 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)
5. Costa, J., Rocha, R.: Global Storing Mechanisms for Tabled Evaluation. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 708–712. Springer, Heidelberg (2008)
6. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Journal of Theory and Practice of Logic Programming 12, 5–34 (2012)
7. Draxler, C.: Accessing Relational and Higher Databases Through Database Set Predicates. PhD thesis, Zurich University (1991)
8. Gray, P.M.D., Lucas, R.J. (eds.): Prolog and Databases, Implementations and New Directions. Ellis Horwood Ltd., Chichester (1988)
9. Gray, P.M.D., Kemp, G.J.L., Rawlings, C.J., Brown, N.P., Sander, C., Thornton, J.M., Orengo, C.M., Wodak, S.J., Richelle, J.: Macromolecular structure information and databases. Trends in Biochemical Sciences 21, 251–256 (1996)
10. Kemp, G.J.L., Iriarte, J.J., Gray, P.M.D.: Efficient Access to FDM Objects Stored in a Relational Database. In: Bowers, D.S. (ed.) BNCOD 1994. LNCS, vol. 826, pp. 170–186. Springer, Heidelberg (1994)
11. Keshava Prasad, T.S., Goel, R., Kandasamy, K., Keerthikumar, S., Kumar, S.: Human protein reference database 2009 update. Nucleic Acids Research 37(suppl. 1), D767–D772 (2009)
12. Lucas, R., Keylink Computers Ltd: ProDBI: ODBC Interface for Quintus Prolog. Keylink Computers Ltd., Kenilworth (1997)
13. Mungall, C.: Experiences using logic programming in bioinformatics. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 1–21. Springer, Heidelberg (2009)
14. Szklarczyk, D., Franceschini, A., Kuhn, M., Simonovic, M., Roth, A., Minguez, P., Doerks, T.: The STRING database in 2011: functional interaction networks of proteins, globally integrated and scored. Nucleic Acids Research 39(suppl. 1), D561–D568 (2011)
15. The UniProt Consortium. Reorganizing the protein space at the universal protein resource (uniprot). Nucleic Acids Res. 40, D71–D75 (2012)
16. Wielemaker, J., Costa, V.S.: On the portability of prolog applications. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 69–83. Springer, Heidelberg (2011)
17. Wielemaker, J., Huang, Z., van der Meij, L.: SWI-Prolog and the Web. Theory and Practice of Logic Programming 8(3), 363–392 (2008)
18. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming 12(1-2), 67–96 (2012)

Dependently Typed Web Client Applications
FRP in Agda in HTML5

Alan Jeffrey

Alcatel-Lucent Bell Labs

Abstract. In this paper, we describe a compiler back end and library for web client application development in Agda, a dependently typed functional programming language. The compiler back end targets ECMAScript (also known as JavaScript), and so is executable in a browser. The library is an implementation of Functional Reactive Programming (FRP) using a constructive variant of Linear-time Temporal Logic (LTL) as its type system.

1 Introduction

Client-side applications are typically model-view-controller architectures, and often include features such as imperative state, concurrency and continuation-passing. These features can result in code which is difficult to reason about, debug and maintain. In this paper, we propose adapting *Functional Reactive Programming* (*FRP*) [13] to the setting of a pure, dependently typed, functional programming language, Agda [1].

Figure 1 shows some simple applications running in a browser. What is interesting about these applications is that they are written in Agda, and compiled to ECMAScript [7]. We have developed a compiler back end, foreign function interface, and library bindings for FRP, and for HTML5 [15] *Document Object Model* (*DOM*) node and event bindings. The compiler extensions have been released as part of Agda 2.3.0, and the libraries are released under an MIT License [4]. Novel features of the compiler and libraries include:

- *Interoperability with ECMAScript idioms.* The compiler makes use of common ECMAScript idioms, to simplify the use of existing ECMAScript libraries in Agda. For example, the Visitor and Observer patterns [14] are used to implement inductive datatypes and notification.
- *Singleton analysis for type erasure.* We perform a static analysis that conservatively approximates singleton types (which have only one inhabitant at run time). Any term of singleton type is replaced by the singleton value at compile time. In particular, we regard `Set` as having singleton value `null`, which allows many type-level computations to be eliminated.
- *View patterns in the FFI.* We support a ECMAScript *Foreign Function Interface* (*FFI*) which, as well as providing bindings for constants and functions, also allows inductive datatypes in Agda to be bound to any ECMAScript type. A variant of view patterns [24] allows pattern-matching to be

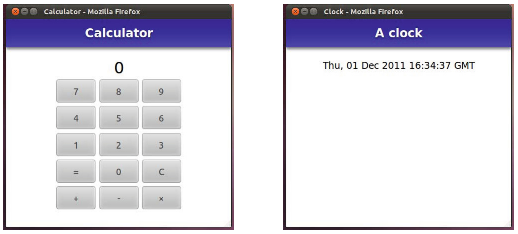


Fig. 1. Example Agda programs running in the browser

compiled to any ECMAScript conditional, for example an Agda boolean type can be compiled to ECMAScript native booleans, without any additional support from the Agda compiler.

- *Linear-time Temporal Logic (LTL) types for FRP.* The semantics of FRP is defined in terms of *signals*, which are time-dependent values. In previous work [16], we showed that signals can be typed using time-dependent types, using the combinators of LTL [23], such that any FRP program is a proof of an LTL tautology.
- *Resource reclamation of FRP signals.* The FRP implementation makes use of techniques from *self-adjusting computation* [8], where signals form a dataflow graph, making use of notifications whenever a signal value changes. We are recording the creation time of each signal in its type, and so can maintain time-sensitive invariants which allow resource reclamation of irrelevant signals, even when the garbage collector regards the signal as still live.
- *Inference of DOM node locations.* A difficult problem in GUI libraries for functional languages is the binding of event listeners to GUI components. In an OO language, binding makes use of object identity, which violates referential transparency since components with identical definitions may have different event streams. In a functional language, this could be modeled by a name creation mechanism [22] or nondeterminism [19], but such models are not compatible with Agda's semantics. We provide a novel form of location inference, which supports the creation of DOM event streams from DOM nodes without violating referential transparency.

Agda is used throughout this paper, but we expect the results would apply to other dependently typed languages, such as Coq [3] or Epigram [5].

Thanks to Sebastian Bocq for detailed comments on this paper.

2 Compiling Agda to ECMAScript

We first consider the design of the ECMAScript back end for the Agda compiler, which is included in Agda 2.3. The compiler translates a dependently typed λ-calculus with inductive datatypes and records into an untyped λ-calculus with records. The interesting features of the compiler are its treatment of singleton

```
data List (A : Set) : Set where
  nil : List A
  cons : A → List A → List A

append : ∀ {A} → List A → List A → List A
append nil bs = bs
append (cons a as) bs = cons a (append as bs)
```

Fig. 2. Example program in Agda

```
data $List\,A$ : Set 0 where {
  $nil : List\,A$,
  $cons : \Pi a\ .\ \Pi as\ .\ List\,A$
}
function $append : \Pi A\ .\ \Pi as\ .\ \Pi bs\ .\ List\,A$
  $= \lambda A\ .\ \lambda as$ . case $as$ of {
    $nil \mapsto \lambda bs\ .\ bs$,
    $cons\ a\ as \mapsto \lambda bs\ .\ cons\ a(append\ A\ as\ bs)$
  }
```

Fig. 3. Example program in Agda IL

```
exports = {
  $nil \mapsto \lambda()\ .\ \lambda(v)\ .\ (v.\ nil())$,
  $cons \mapsto \lambda(a, as)\ .\ \lambda(v)\ .\ (v.\ cons(a, as))$,
  $append \mapsto \lambda(A)\ .\ \lambda(as)\ .\ (as(\{$
    $nil \mapsto \lambda()\ .\ \lambda(bs)\ .\ bs$,
    $cons \mapsto \lambda(a, as)\ .\ \lambda(bs)$ . (exports. $cons(a,$ exports. $append(A)(as)(bs)))$
  })
}
```

Fig. 4. Example program in ECMAScript IL

```
define(["exports",function(exports) {
  exports.nil = function() { return function(v) { return v.nil(); }; };
  exports.cons = function(a,as) { return function(v) { return v.cons(a,as); }; };
  exports.append = function(A) { return function(as) { return as({
    nil: function() { return function(bs) { return bs; }; },
    cons: function(a,as) { return function(bs) {
      return exports.cons(a,exports.append(A)(as)(bs));
    }; }
  }); }; };
});
```

Fig. 5. Example program in ECMAScript

types (including type erasure, since Set is treated as a singleton type) and the translation of datatypes to a use of the visitor pattern.

In Figures 2–5, we show how a simple datatype and recursive function (append over lists) is translated first into an *Agda Intermediate Language* (*IL*), then an *ECMAScript IL*, and finally into ECMAScript:

- The translation from Agda (Figure 2) to Agda IL (Figure 3) is not novel, and handles issues such as making implicit arguments explicit and η-normalizing function applications. In this paper, we give a presentation using case statements in the IL rather than pattern matching. We compile pattern matches to case using decision trees (credited by Cardelli [11] to Kahn and MacQueen in the HOPE compiler [10]).
- The translation from the Agda IL (Figure 3) to the ECMAScript IL (Figure 4) is the interesting one, and is discussed in more detail below. Note that in this translation, case statements over inductive datatypes have been replaced by uses of the visitor pattern, and that top-level declarations in an Agda module have been replaced by fields in an ECMAScript record exports.
- The translation from the ECMAScript IL (Figure 4) to ECMAScript (Figure 5) is routine. We make use of the *Asynchronous Module Definition* (*AMD*) [2] module system for ECMAScript, which supports a special object exports. The translation of an Agda module is an ECMAScript module which assigns to the appropriate exports field.

Figure 6 shows a simplified grammar for the Agda IL, which is a λ-calculus with records, inductive datatypes, Π types, stratified Set types and postulates (uninterpreted constants). The main differences between this presentation of the IL and the actual implementation are modules, namespacing, type information, and the use of case expressions rather than pattern matching functions.

Figure 7 shows a simplified grammar for the ECMAScript IL, which is an untyped λ-calculus with records. The main differences between this presentation of the IL and the actual implementation are namespacing, conditionals and infix and prefix operators. Note that many features of ECMAScript are missing from the ECMAScript IL, such as mutable state, prototypes and constructors. The ECMAScript IL allows importing arbitrary AMD modules, so these features can still be used, as long as they are in an imported module.

We define β-reduction as per usual in a λ-calculus with records. The only point of interest in the definition is the use of undef in ECMAScript's semantics. For example, we define capture-avoiding substitution $M[\vec{N}/\vec{x}]$ in the usual way whenever $|\vec{N}| = |\vec{x}|$, then generalize to arbitrary $\vec{N}$ and $\vec{x}$ by substituting undef if necessary:

$$\begin{aligned} M[(\vec{N},\vec{L})/\vec{x}] &= M[\vec{N}/\vec{x}] && \text{when } |\vec{N}| = |\vec{x}| \\ M[\vec{N}/(\vec{x},\vec{y})] &= M[\vec{N}/\vec{x}, \mathsf{undef}/\vec{y}] && \text{when } |\vec{N}| = |\vec{x}| \end{aligned}$$

from which we define β-reduction of functions:

$$(\lambda(\vec{x}) \,.\, M)(\vec{N}) \rightarrow M[\vec{N}/\vec{x}]$$

$$\begin{aligned} A, B, C &::= x\,\vec{A} \mid \lambda x \,.\, A \mid \{\vec{\ell} \mapsto \vec{B}\} \mid A.\ell \mid g\,\vec{A} \mid k \mid \\ &\quad \mid \Pi x \,.\, A \mid \mathsf{Set}\ A \mid c\,\vec{A} \mid \mathsf{case}\,A\,\mathsf{of}\ \{\vec{P} \mapsto \vec{B}\} \\ P, Q &::= c\,\vec{x} \\ D, E &::= \mathsf{function}\,g : A = B \mid \mathsf{data}\,g\,\vec{x} : B\ \mathsf{where}\ \{\vec{c} : \vec{C}\} \mid \\ &\quad \mathsf{record}\,g\,\vec{x} : B\ \mathsf{where}\ \{\vec{\ell} : \vec{C}\} \mid \mathsf{postulate}\,g : A \end{aligned}$$

Fig. 6. Agda IL

$$L, M, N ::= x \mid \lambda(\vec{x}) \,.\, M \mid M(\vec{N}) \mid \{\,\vec{\ell} \mapsto \vec{M}\,\} \mid M.\ell \mid k$$

Fig. 7. ECMAScript IL

Similarly, field access of an object returns undef for missing fields:

$$\{\,\vec{\ell} \mapsto \vec{M}\,\}.\ell \to \begin{cases} M_i & \text{if } \ell = \ell_i \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

In Figures 8–9 we show how Agda IL is translated into ECMAScript IL. Most of the translation is direct, but there are two points of interest: a static approximation of *singleton types*, and the *visitor pattern* [14] for inductive datatypes.

For singleton types, we include a judgement "A has singleton B", meaning that any closed instance of type A must be equal to B. For example:

- $\top$ (a record type with no fields) has singleton $\{\,\}$.
- $\bot$ (an inductive type with no constructors) has no closed instances, so we can declare that $\bot$ has singleton undef. Since $\neg A$ is defined to be $A \to \bot$, it has singleton $\lambda x.\mathsf{undef}$, and so we can eliminate many instances of negations.
- Since we are using a type-erasing translation, Set A (the type of types at universe level A) has singleton null. This eliminates many instances of run-time type computation.

The visitor pattern uses double callbacks to emulate case statements (in [9], this form of visitor is called an *external* visitor, in contrast to an *internal* visitor which emulates a recursion scheme). For example, if as is a list, then:

```
as({ nil: f, cons: g })
```

will call back f() if as is an empty list, and g(b,bs) if as has head b and tail bs. The translation of case statements into visitors is direct.

Recall that recursive declarations are translated to imperative updates to the mutable exports variable. For mutually recursive declarations under a λ (such as the traditional *even* and *odd* functions) this presents no problem, but for top-level recursive declarations, we have to ensure that exports are defined before

$$\begin{aligned}
[\![x\,\vec{A}]\!] &= x([\![A_1]\!])\cdots([\![A_n]\!])\\
[\![\lambda x\,.\,A]\!] &= \lambda x\,.\,[\![A]\!]\\
[\![\{\vec{\ell}\mapsto\vec{A}\}]\!] &= \{\,\vec{\ell}\mapsto[\![\vec{A}]\!]\,\}\\
[\![A.\ell]\!] &= [\![A]\!].\ell\\
[\![g\,\vec{A}]\!] &= \begin{cases} C & \text{if } g\,\vec{A} : B \text{ and } B \text{ has singleton } C\\ \mathsf{exports}.g\,[\![\vec{A}]\!] & \text{otherwise}\end{cases}\\
[\![k]\!] &= k\\
[\![\Pi x\,.\,A]\!] &= \mathsf{null}\\
[\![\mathsf{Set}\ A]\!] &= \mathsf{null}\\
[\![c\,\vec{A}]\!] &= c([\![\vec{A}]\!])\\
[\![\mathsf{case}\,A\,\mathsf{of}\ \{\vec{P}\mapsto\vec{B}\}]\!] &= A(\{\,[\![\vec{P}\mapsto\vec{B}]\!]\,\})\\
[\![c\,\vec{x}\mapsto B]\!] &= c\mapsto\lambda(\vec{x})\,.\,[\![B]\!]
\end{aligned}$$

$$\frac{\mathsf{data}\,g\,\vec{x} : \mathsf{Set}\ A\ \mathsf{where}\ \{\ \}}{g\,\vec{A}\text{ has singleton }\mathsf{undef}}\qquad\frac{\mathsf{data}\,g\,\vec{x} : \mathsf{Set}\ A\ \mathsf{where}\ \{c : g\,\vec{x}\}}{g\,\vec{A}\text{ has singleton }c}\qquad\frac{\mathsf{record}\,g\,\vec{x} : \mathsf{Set}\ A\ \mathsf{where}\ \{\ \}}{g\,\vec{A}\text{ has singleton }\{\ \}}$$

$$\frac{A\text{ has singleton }B}{\Pi x\,.\,A\text{ has singleton }\lambda x\,.\,B}\qquad\frac{}{\mathsf{Set}\ A\text{ has singleton }\mathsf{null}}$$

Fig. 8. Translation of Agda expressions to ECMAScript

$$\begin{aligned}
[\![\mathsf{function}\,g : A = B]\!] &= g\mapsto\begin{cases} C & \text{if } A \text{ has singleton } C\\ [\![A]\!] & \text{otherwise}\end{cases}\\
[\![\mathsf{data}\,g\,\vec{x} : A\ \mathsf{where}\ \{\vec{c} : \vec{B}\}]\!] &= [\![\vec{c} : \vec{B}]\!]\\
[\![\mathsf{record}\,g(\vec{x} : \vec{A}) : B\ \mathsf{where}\ \{\vec{\ell} : \vec{C}\}]\!] &= \epsilon\\
[\![\mathsf{postulate}\,g : A]\!] &= g\mapsto\begin{cases} B & \text{if } A \text{ has singleton } B\\ \mathsf{undef} & \text{otherwise}\end{cases}\\
[\![c : \Pi\vec{x}\,.\,g\,\vec{A}]\!] &= c\mapsto\lambda(\vec{x})\,.\,\lambda(v)\,.\,v.c(\vec{x})
\end{aligned}$$

Fig. 9. Translation of Agda declarations to ECMAScript

their use. Consider the Agda IL declaration:

$$\mathsf{function}\,x : \mathbb{N} = y + 1 \quad \mathsf{function}\,y : \mathbb{N} = 3$$

and then translated into ECMAScript IL it is:

$$\{\,x\mapsto\mathsf{exports}.y + 1; y\mapsto 3\,\}$$

Unfortunately, translated directly into ECMAScript, this would be:

```
define(["exports"],function(exports) {exports.x = exports.y + 1;exports.y = 3;});
```

which generates a load-time error, since exports.y is undefined at the point of its use. To avoid this, we inline any occurrences of exports which occur outside of an enclosing λ. In this example, inlining exports produces:

$$\{\, x \mapsto \{\, x \mapsto \mathsf{exports}.y + 1; y \mapsto 3 \,\}.y + 1; y \mapsto 3 \,\}$$

which β-reduces to:

$$\{\, x \mapsto 3 + 1; y \mapsto 3 \,\}$$

and translates into ECMAScript as:

```
define(["exports"],function(exports) {exports.x = 3 + 1;exports.y = 3;});
```

As this example shows, we use a nïave strategy of always inlining top-level occurrences of exports. Since Agda is total, this process must terminate (unless Agda's termination checker is disabled, in which case the compiler is not guaranteed to terminate) but may result in an exponential blowup in program resource usage. We leave a more sophisticated treatment of inlining for future work.

3 Foreign Function Interface

For Agda to be useful for writing web applications, it must interact with native ECMAScript APIs, notably those defined by HTML5. The translation of Agda into ECMAScript is designed to make this as simple as possible (for example, translating functions to functions, and records to records) but there is still a need for a *Foreign Function Interface* (*FFI*) to provide bindings for native types. In Agda, FFIs are defined via pragmas, for example to bind Agda identifier g to ECMAScript term M:

COMPILED_JS g M

In the case of functions, constructors or postulates, the semantics of FFI code is direct: the ECMAScript is inlined (and β-reduced) whenever the identifier is used. For example if we define:

```
data ℕ : Set where      _+_ : ℕ → ℕ → ℕ           _*_ : ℕ → ℕ → ℕ
  zero : ℕ              zero  + y = y             zero  * y = zero
  suc : ℕ → ℕ           suc x + y = suc (x + y)   suc x * y = y + (x * y)
```

then the following pragma declarations bind zero, suc, + and * to their native counterparts:

```
COMPILED_JS zero 0
COMPILED_JS suc function(x) { return x+1; }
COMPILED_JS _+_ function(x) { return function(y) { return x+y; }; }
COMPILED_JS _*_ function(x) { return function(y) { return x*y; }; }
```

By itself, however, this is not sufficient, as user code may include recursive functions over naturals, such as the ever-popular factorial:

```
fact : ℕ → ℕ
fact zero = suc zero
fact (suc x) = suc x * fact x
```

To support this, we allow FFI bindings from datatypes to the *acceptor function* for that datatype, similar to *view patterns* [24]. The acceptor is a function $f(x, v)$ which takes as parameters a value x, and a visitor v, and calls the appropriate visitor method. For example if we declare:

```
COMPILED_JS ℕ function (x,v) {
  if (x < 1) { return v.zero(); } else { return v.suc(x-1); }
}
```

then the generated ECMAScript for the factorial function is:

```
exports.fact = function (x) {
  if (x < 1) { return 0+1; } else { return ((x-1)+1) * exports.fact(x-1); }
}
```

4 Functional Reactive Programming

The style of programming used in web applications such as those in Figure 1 is *Functional Reactive Programming* (*FRP*) [13]. The semantics of FRP is defined in terms of *signals*, which are thought of as time-dependent values. For example, the clock application is:

$$\mathsf{main} = \mathsf{text}(\mathsf{map}\,\mathsf{toUTCString}(\mathsf{every}(1\,\mathsf{sec})))$$

where:

- $\mathsf{every}(1\,\mathsf{sec})$ is a signal of Time, which updates every second,
- $\mathsf{map}\,f(\sigma)$ applies a function $f : A \to B$ to a signal σ of A to get a signal of B, here $\mathsf{toUTCString} : \mathsf{Time} \to \mathsf{String}$, and
- $\mathsf{text}(\sigma)$ converts a signal σ of String to a signal of DOM nodes.

The types of these combinators are (ignoring some issues about the type for DOM nodes, which we return to in Section 6):

$$\begin{aligned}
\mathsf{every} &: \mathsf{Delay} \to [\![\Box\langle\mathsf{Time}\rangle]\!] \\
\mathsf{map} &: [\![A \Rightarrow B]\!] \to [\![\Box A \Rightarrow \Box B]\!] \\
\mathsf{text} &: [\![\Box\langle\mathsf{String}\rangle \Rightarrow \Box\mathsf{DOM}]\!]
\end{aligned}$$

which gives the type of main as $[\![\Box\mathsf{DOM}]\!]$, that is a signal of DOM nodes, suitable for rendering in a browser.

These types are based on *Linear-time Temporal Logic* (*LTL*) [23]. In previous work [16] we showed that FRP programs in a dependently typed programming language can be given types in a constructive variant of LTL, such that any well-typed FRP program is a proof of an LTL tautology. The correspondence between FRP programs and LTL proofs was discovered independently by Jeltsch [18]. Since LTL propositions are parameterized over time, we consider types parameterized over time, that is *reactive types*:

$$\mathsf{RSet} = \mathsf{Time} \to \mathsf{Set}$$

where Time is a totally ordered set (implemented using ECMAScript's time model, which is an integer number of milliseconds since 1 Jan 1970). Some combinators on reactive types are:

$$\begin{array}{rl}
\langle\cdot\rangle : \mathsf{Set} \to \mathsf{RSet} & \langle A\rangle = \lambda t \,.\, A \\
(\cdot \Rightarrow \cdot) : \mathsf{RSet} \to \mathsf{RSet} \to \mathsf{RSet} & A \Rightarrow B = \lambda t \,.\, A(t) \to B(t) \\
[\![\cdot]\!] : \mathsf{RSet} \to \mathsf{Set} & [\![A]\!] = \forall\{t\} \,.\, A(t) \\
\Box : \mathsf{RSet} \to \mathsf{RSet} & \Box A = \,?
\end{array}$$

These combinators are:

- $\langle A\rangle$ is a constant reactive type; viewed as a temporal proposition $\langle A\rangle$ is true at time t when A is true.
- $A \Rightarrow B$ is the pointwise function space between A and B; viewed as a temporal proposition, $A \Rightarrow B$ is true at time t when A being true at time t implies that B is true at time t.
- $[\![A]\!]$ embeds RSet back into Set; viewed as a proposition $[\![A]\!]$ is true whenever A is a tautology, that is A is true at all times t.
- $\Box A$ is the type of signals of A; viewed as a temporal proposition, $\Box A$ is true at time t whenever A is true at any time $u \geq t$.

The FRP combinators can be viewed as a proof system for LTL, for example one of the axioms of S4 modal logic is given by:

$$\mathsf{map} : [\![A \Rightarrow B]\!] \to [\![\Box A \Rightarrow \Box B]\!]$$

Note that we do *not* give a definition for $\Box A$ (the library defines it as a postulate). It is isomorphic to LTL's "global" modality:

$$\Box A \approx \lambda t \,.\, \forall u \,.\, \mathsf{True}(t \leq u) \to A\,u$$

where:

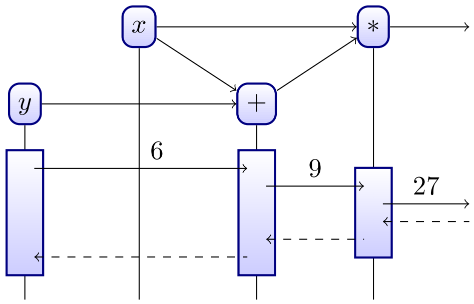
$$\begin{aligned}
\mathsf{True}(\cdot) \ &: \ \mathsf{Bool} \to \mathsf{Set} \\
\mathsf{True}(b) &= \begin{cases} 1 \text{ if } b = \mathsf{true} \\ 0 \text{ otherwise} \end{cases}
\end{aligned}$$

The implementation of $\Box A$ in ECMAScript is not given functionally. Instead it is given as a dataflow graph, where the nodes implement the observer pattern [14]. The implementation is based on *self-adjusting computation* [8,20], and is similar to FrTime [12], Flapjax [21] and Froc [6].

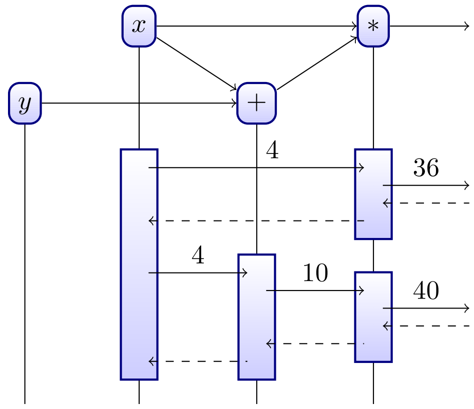
Consider the dataflow graph for the expression $x * (y + x)$:

$$\begin{tikzcd}
& \boxed{x} \arrow[rr] \arrow[dr] & & \boxed{*} \arrow[r] & {} \\
\boxed{y} \arrow[rr] & & \boxed{+} \arrow[ur] & &
\end{tikzcd}$$

This is implemented as an object graph, where every node implements the Observer pattern, and memoizes its current value. When an update takes place (for example, an external event arrives) the nodes send notifications to their obervers, requesting that they update themselves, and recursively inform their observers if necessary. Note that nodes only notify their observers when their values change, so unchanged nodes are not involved in any updates. For example, if x's value is 3, and y's value is updated to 6 then the following notifications are sent:



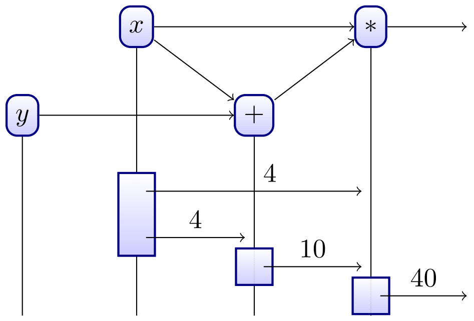
Unfortunately, a simple application of the observer pattern results in *glitches*. These are transient erroneous values, due to nodes receiving multiple notifications. For example, if x's value is updated to 4 then the following notifications could be sent:



In this example, the $*$ node has been notified twice, and as result has sent two notifications, the first of which does not match the FRP semantics. To avoid such glitches, we adopt the same strategy as [12,21], and *rank* nodes, such that every node has smaller rank than all of its observers[1]. Notifications are now

[1] Acar [8] uses post-order traversal order rather than height order, because his target languages allow for exceptions and other error behaviours, and so (for example) conditionals must be evaluated before branches in an if-expression. Agda is total, and so we can use a simpler ordering strategy, at the possible cost of wasted effort.

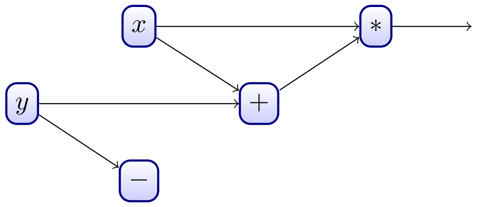
asynchronous rather than synchronous, and are executed in rank order. For example, the above glitchy behaviour is replaced by:



For efficiency, we use synchronous notification for nodes with fan-in one, and reserve asynchronous notification for nodes with fan-in of two or more. The implementation of flow graphs makes use of a task scheduler, which handles asynchronous notification, and ensures that notifications are processed in rank order. The scheduler also supports delayed notification (using the HTML5 time-out mechanism) and rank updates (a node may *switch* its observed neighbours, causing its rank to change, which must be propagated upwards).

5 Garbage Collecting FRP

Nodes in the flow graph of an FRP program maintain a set of observer nodes (which should be notified on state updates) and observed nodes (whose memoized values can be queried when a notification is processed). This presents a challenge to a garbage collector, since bidirectional links may keep nodes alive unnecessarily. For example, consider the graph:



Here there is a node $-$ with no observers, which should be reclaimed. Such unobserved nodes can arise dynamically due to switches, which reconfigure the node graph. To reclaim unobserved nodes, some FRP implementations [12] make use of *weak pointers* for observers, which would allow garbage collection in this case. Unfortunately, ECMAScript does not support weak pointers.

An alternative is to have the FRP library handle node reclamation. Nodes have addObserver and removeObserver methods: when a node has its last observer removed, it calls removeObserver on each of its observed neighbours to remove itself. Essentially, this is a reference counting garbage collector (cyclic flow graphs are handled by an explicit fixed point function, which does not increase the reference count, so cycles can be collected). These functions are only visible in ECMAScript: they have mutable semantics, so must be kept hidden from Agda.

Unfortunately, this is not always safe, since a node might be added back into the graph after it has been reclaimed:

```
// node1 starts out with just observer node2
node1.removeObserver(node2);
// at this point node1 reclaims its resources
node1.addObserver(node3);
```

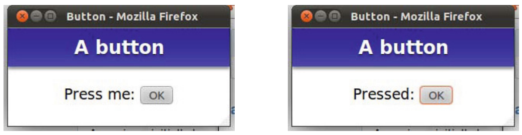
Without some additional guarantees, node1 would reclaim its resources, only later to be added back into the flow graph in an unsafe state. To avoid this, we maintain two invariants:

1. the state of a node is only ever queried by its observers, and
2. a node only ever has observers added during the time slice that it is created.

In the presence of these invariants, we have a safe variant of removeObserver: when a node has its last observer removed *and we have finished processing the time slice that created the node*, it calls removeObserver on each of its observed neighbours to remove itself. In ECMAScript, there is no way to statically enforce the invariants, but in Agda we can do this because the LTL type for a signal $\Box A(t)$ carries a time parameter t which records its start time. The API for signals only allows signals to be built at their start time (for example map f converts a signal of type $\Box A(t)$ to a signal of type $\Box B(t)$, that is the start time is preserved). This technique for tracking creation times is similar to Jeltsch's *era* parameters [17]. Since Agda is a dependent language, we can embed start times directly in types, rather than having to use phantom types for this purpose.

6 Bindings for DOM Nodes and Events

In Figure 1 we showed a calculator application, built in Agda. A prototypical example of a GUI is a single button:



The source for this program is quite simple:

```
main = lab ++ but where
  but = element "button"(text(const "OK"))
  clk = listen click but
  lab = text(hold "Press me: " (tag "Pressed: " clk))
```

This declares a button but, and then some text lab whose value depends on the stream clk of click events coming from but. The boilerplate $\mathsf{hold}\,x\,(\mathsf{tag}\,y\,\sigma)$ is a behaviour which starts as value x, and switches to y after the first event from σ.

The types of the functions used here are (somewhat simplified):

$$\begin{aligned}
* &: \mathsf{RSet} \to \mathsf{RSet}\\
\mathsf{Mouse} &: \mathsf{RSet}\\
\mathsf{EventType} &: \mathsf{RSet} \to \mathsf{Set}\\
\mathsf{click} &: \mathsf{EventType}\,\mathsf{Mouse}\\
\mathsf{listen} &: \forall\{A\} \to \mathsf{EventType}\,A \to [\![\square\mathsf{DOM} \Rightarrow *A]\!]\\
\mathsf{const} &: \forall\{A\} \to [\![A]\!] \to [\![\square A]\!]\\
\mathsf{tag} &: \forall\{A\,B\} \to [\![B]\!] \to [\![*A \Rightarrow *B]\!]\\
\mathsf{hold} &: \forall\{A\} \to [\![\langle A\rangle \Rightarrow *\langle A\rangle \Rightarrow \square\langle A\rangle]\!]\\
\mathsf{element} &: \mathsf{String} \to [\![\square\mathsf{DOM} \Rightarrow \square\mathsf{DOM}]\!]\\
(\cdot +\!\!+ \cdot) &: [\![\square\mathsf{DOM} \Rightarrow \square\mathsf{DOM} \Rightarrow \square\mathsf{DOM}]\!]
\end{aligned}$$

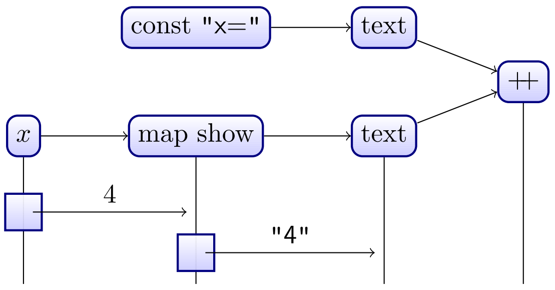
Here:

- $*A$ is the reactive type of event streams, where any event at time t has type $A(t)$. It is implemented in a similar fashion to $\square A$.
- Mouse is the reactive type of mouse events.
- $\mathsf{EventType}\ A$ is the type of codes for events of type A, for example click is a code for events of type Mouse.
- $\mathsf{listen}\,c\,\sigma$ is an event stream which listens for events with code c coming from DOM nodes σ. For example, $\mathsf{listen}\,\mathsf{click}\,\mathsf{but}$ is the stream of click events coming from the button but.
- $\mathsf{const}\,x$ is a constant signal that always returns x.
- $\mathsf{tag}\,x\,\sigma$ is a stream of x events which fires whenever σ fires.
- $\mathsf{hold}\,x\,\sigma$ converts an event stream to a signal, by returning the most recent value from σ (or x if there is none).
- $\mathsf{element}\,a\,\sigma$ constructs a DOM node with tag a and children σ.
- $\sigma +\!\!+ \tau$ concatenates the DOM nodes from σ with those from τ.

The implementation of these functions is fairly straightforward ECMAScript programming using the HTML5 API, for example $\mathsf{Mouse}(t)$ is inhabited by mouse events processed at time t, and $\mathsf{listen}\,\mathsf{click}\,\sigma$ registers a click event handler to each DOM node generated by σ (and deregisters them when the event stream has no observers). DOM nodes are sinks for notifications, for example in the program:

```
text (const "x=") ++ text (map show x)
```

if x's value is updated to 4 then the following notifications are sent; note that the text node does *not* update its parent:



The library behaves as expected if a user declares multiple buttons, for example:

$$\begin{aligned}
&\mathsf{main} = \mathsf{lab} \mathbin{+\!\!+} \mathsf{but}_1 \mathbin{+\!\!+} \mathsf{but}_2\ \mathsf{where}\\
&\quad \mathsf{but}_1 = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"}))\\
&\quad \mathsf{but}_2 = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"}))\\
&\quad \mathsf{clk} = \mathsf{listen}\,\mathsf{click}\,\mathsf{but}_1\\
&\quad \mathsf{lab} = \mathsf{text}(\mathsf{hold}\,\texttt{"Press me: "}\,(\mathsf{tag}\,\texttt{"Pressed: "}\,\mathsf{clk}))
\end{aligned}$$

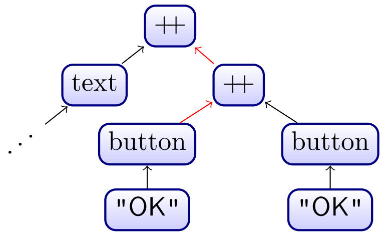
only changes the text when the first button is pressed, not the second, since it only listens to but_1 and not but_2. On the surface, this appears to violate Agda's semantics, which includes β-equivalence, and hence referential transparency. It appears that but_1 and but_2 have the same definition, but yet the behaviour depends on which button we listen to.

Krishnaswami and Benton [19] resolve this by giving event streams a nondeterministic semantics, which the implementation is given freedom to resolve in any way it likes. In practice, the implementation uses node identity to resolve nondeterminism. Since the semantics is nondeterministic, it is no longer defined in a cartesian closed category of sets and functions, but instead in the monoidal closed category of sets and relations. Krishnaswami and Benton provide a DSL with a linear type system for writing such nondeterministic programs. In our GUI library, we are using Agda's native function space to express reactive programs, so we cannot use a nondeterministic semantics.

In fact, the above example does not violate referential transparency, and instead is using implicit arguments to name components. Above, we noted that we had simplified the presentation of the types for DOM nodes. In fact, we do not have $\mathsf{DOM} : \mathsf{RSet}$, instead we have:

$$\mathsf{DOM} : \mathsf{DOW} \to \mathsf{RSet}$$

where DOW is a type of *Document Object Worlds* (or "upside-down DOMs"). A value of type DOW records *where* in a DOM tree a node lives, for example in the DOM flow graph:



the route from the the first button node to the root node is left, then right, which we write as $\mathsf{left}(\mathsf{right}(\ell))$ where ℓ is the location of the root node. DOWs are postulated as:

$$\begin{aligned}\mathsf{DOW} &: \mathsf{Set}\\ \mathsf{left}, \mathsf{right} &: \mathsf{DOW} \to \mathsf{DOW}\\ \mathsf{child} &: \mathsf{String} \to \mathsf{DOW} \to \mathsf{DOW}\end{aligned}$$

Under the hood, a DOW is implemented as a container of DOM nodes, with pointers to all of the DOM nodes at that location (typically there is just one, but since Agda does not support linear types there is no way to enforce that convention).

We can now reveal the "real" types for the DOM-manipulating functions:

$$\begin{aligned}\mathsf{text} &: \forall\{\ell\} \to [\![\Box\langle\mathsf{String}\rangle \Rightarrow \Box(\mathsf{DOM}\,\ell)]\!]\\ \mathsf{element} &: \forall a\,\{\ell\} \to [\![\Box(\mathsf{DOM}(\mathsf{child}\ a\ \ell)) \Rightarrow \Box(\mathsf{DOM}\,\ell)]\!]\\ (\cdot +\!\!+ \cdot) &: \forall\{\ell\} \to [\![\Box(\mathsf{DOM}(\mathsf{left}\ \ell)) \Rightarrow \Box(\mathsf{DOM}(\mathsf{right}\ \ell)) \Rightarrow \Box(\mathsf{DOM}\ \ell)]\!]\\ \mathsf{listen} &: \forall\{A\ \ell\} \to \mathsf{EventType}\,A \to [\![\Box\mathsf{DOM}\ \ell \Rightarrow *A]\!]\end{aligned}$$

For example, we can make the inferred types explicit in our problematic example:

$$\begin{aligned}&\mathsf{main} : \forall\{\ell\} \to [\![\Box(\mathsf{DOM}\ \ell)]\!]\\ &\mathsf{main}\{\ell\}\{t\} = \mathsf{lab} +\!\!+ \mathsf{but}_1 +\!\!+ \mathsf{but}_2\ \mathsf{where}\\ &\quad \mathsf{but}_1 : \Box(\mathsf{DOM}(\mathsf{left}(\mathsf{right}(\ell))))\ t\\ &\quad \mathsf{but}_1 = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"}))\\ &\quad \mathsf{but}_2 : \Box(\mathsf{DOM}(\mathsf{right}(\mathsf{right}(\ell))))\ t\\ &\quad \mathsf{but}_2 = \mathsf{element}\,\texttt{"button"}(\mathsf{text}(\mathsf{const}\,\texttt{"OK"}))\\ &\quad \mathsf{clk} : *\mathsf{Mouse}\ t\\ &\quad \mathsf{clk} = \mathsf{listen}\{\mathsf{Mouse}\}\{\mathsf{left}(\mathsf{right}(\ell))\}\mathsf{click}\,\mathsf{but}_1\\ &\quad \mathsf{lab} : \Box(\mathsf{DOM}(\mathsf{left}(\ell)))\ t\\ &\quad \mathsf{lab} = \mathsf{text}(\mathsf{hold}\,\texttt{"Press me: "}\,(\mathsf{tag}\,\texttt{"Pressed: "}\,\mathsf{clk}))\end{aligned}$$

With the optional arguments in place, we can see how referential transparency is being maintained: the optional argument to listen is $\mathsf{left}(\mathsf{right}(\ell))$, which is why the value of lab depends on but_1 being pressed but not but_2. Agda's ability to infer expressions as well as types is being used to provide a referentially transparent semantics to a program which looks like it depends on object identity.

References

1. The Agda wiki, http://wiki.portal.chalmers.se/agda/
2. Asynchronous module definition API, https://github.com/amdjs/
3. The Coq proof assistant, http://coq.inria.fr/
4. ECMAScript back end for functional reactive programming in Agda, https://github.com/agda/agda-frp-js
5. The Epigram 2 programming language, http://www.e-pig.org/darcs/Pig09/web/
6. Froc: Functional reactive programming in O'Caml, https://jaked.github.com/froc/
7. ECMAScript language specification. ECMA Standard 262, 5.1 Edn. (2011)
8. Acar, U.A.: Self-Adjusting Computation. PhD thesis, Carnegie Mellon Univ. (2005)
9. Buchlovsky, P., Thielecke, H.: A type-theoretic reconstruction of the visitor pattern. In: Proc. Mathematical Foundations of Programming Semantics, pp. 309–329 (2006)
10. Burstall, R.M., MacQueen, D.B., Sannella, D.: HOPE: An experimental applicative language. In: Proc. LISP Conf., pp. 136–143 (1980)
11. Cardelli, L.: Compiling a functional language. In: Proc. ACM Symp. LISP and Functional Programming, pp. 208–217 (1984)
12. Cooper, G.H., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 294–308. Springer, Heidelberg (2006)
13. Elliott, C., Hudak, P.: Functional reactive animation. In: Proc. Int. Conf. Functional Programming, pp. 263–273 (1997)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
15. Hickson, I., et al.: HTML5: A vocabulary and associated APIs for HTML and XHTML. W3C Working Draft (2011), http://www.w3.org/TR/html5/
16. Jeffrey, A.S.A.: LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In: Proc. ACM Workshop Programming Languages Meets Program Verification (2012)
17. Jeltsch, W.: Signals, not generators! In: Proc. Symp. Trends in Functional Programming, pp. 283–297 (2009)
18. Jeltsch, W.: The Curry-Howard correspondence between temporal logic and functional reactive programming (2011), http://www.cs.ut.ee/~varmo/tday-nelijarve/jeltsch-slides.pdf
19. Krishnaswami, N., Benton, N.: A semantic model for graphical user interfaces. In: Proc. ACM Int. Conf. Functional Programming, pp. 45–57 (2011)
20. Ley-Wild, R.: Programmable Self-Adjusting Computation. PhD thesis, Carnegie Mellon Univ. (2010)
21. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for Ajax applications. In: Proc. ACM Conf. Object Oriented Programming Systems Languages and Applications, pp. 1–20 (2009)
22. Pitts, A.M., Stark, I.D.B.: Observable properties of higher order functions that dynamically create local names, or: What's new? In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 122–141. Springer, Heidelberg (1993)
23. Pnueli, A.: The temporal logic of programs. In: Proc. Symp. Foundations of Computer Science, pp. 46–57 (1977)
24. Wadler, P.: Views: a way for pattern matching to cohabit with data abstraction. In: Proc. ACM Symp. Principles of Programming Languages, pp. 307–313 (1987)

Parallel Performance of Declarative Programming Using a PGAS Model

Rui Machado[1,2], Salvador Abreu[2], and Daniel Diaz[3]

[1] Fraunhofer ITWM, Kaiserslautern, Germany
rui.machado@itwm.fhg.de
[2] Universidade de Évora and CENTRIA, Portugal
spa@di.uevora.pt
[3] University of Paris 1-Sorbonne, France
Daniel.Diaz@univ-paris1.fr

Abstract. Constraint Programming is one approach to declarative programming where a problem is modeled as a set of variables with a domain and a set of relations (constraints) between them. Constraint-based Local Search builds on the idea of using constraints to describe and control local search. Problems are modeled using constraints and heuristics for which solutions are searched, using Local Search. With the progressing move toward multi and many-core systems, parallelism has become mainstream as the number of cores continues to increase. Declarative programming approaches such as those based on constraints need to be better understood and experimented in order to understand their parallel behaviour. In this paper, we discuss experiments we have been carrying out with Adaptive Search and present a new parallel version of it based on GPI, a recent API and programming model for the development of scalable parallel applications. Our experiments on different problems show interesting speed-ups and, more importantly, a better understanding of how these gains are obtained, in the context of declarative programming.

Keywords: Constraint Programming, Local Search, Parallel Programming.

1 Introduction

There is an inevitable paradigm shift towards multicore technologies where parallelism is now omnipresent. In recent systems, parallelism spreads over several systems levels and heterogeneity is growing on the node as well as on the chip level. Data must be maintained across a hierarchy of memory levels and most applications and algorithms are not ready to take full advantage of the available capabilities.

Parallel programming is usually a difficult and error-prone task. Although MPI [11] has become the *de facto* standard for parallel programming, there has been a demand for programming models with a flexible threads model and asynchronous communication. PGAS (Partitioned Global Address Space) programming models have been emerging as a valid alternative to MPI.

One of the great features of declarative programming approaches is their potential simplification of the development of parallel programs, relieving the programmer from error-prone aspects related to explicit control, which can be very difficult to handle with parallel programming, while retaining enough expressive power to model complex real-world problems. One declarative approach is *Constraint Programming*: a problem is modeled as a set of variables over some domain and a set of relations (constraints) is required to hold between them. Program execution consists in finding a solution to (i.e., solving) the stated constraint problem. The solving process can use different methods, one of which is *Local Search* where, instead of exploring the complete search space, heuristics are used to guide the search to portions of the search space where solutions are more likely to be found. Local Search is based on the simple idea of "searching" by iteratively moving from one candidate solution to one of its *neighbours*. Despite its simplicity and effectiveness to handle hard problems, in order to solve large problem instances, parallelism should be introduced to cope with the large running time.

Our general aim is not only to simplify the use of parallelism of current systems with a declarative approach based on constraints but, at the same time, exploit that parallelism to tackle large and difficult problems.

We have been developing parallel designs for both complete (propagation-based [14]) and local search constraint solvers. This article reports on the latter.

The contribution of this paper is twofold: a new parallel design for the Adaptive Search method based on a PGAS Model and a better understanding of its parallel behaviour, easily extended to Local Search algorithms in general. We present and evaluate our new design based on GPI, showing interesting speed-up gains on benchmarks known to have scalability issues. We discuss the results and provide a deeper interpretation of the parallel behaviour of Adaptive Search in particular and of Local Search methods in general, based on some characteristics of the benchmarks.

The rest of the paper is organized as follows: in section 2 we present GPI and its programming model, hightlighting its important features. Section 3 provides some background on the Adaptive Search algorithm and section 4 focuses on its parallelization. In section 5, we detail our parallelization strategy based on GPI and in section 6 we show the obtained results and compare it to the previous implementation. Section 7 examines and interprets our experimental findings, correlating them with the characteristics of the problems. Finally, section 8 presents a short conclusion and perspectives of future work.

2 GPI

The Partitioned Global Address Space (PGAS) is a parallel programming model which has been seen as a good alternative to the established MPI. The PGAS approach offers the programmer an abstract shared address space model which simplifies the programming task and at the same time facilitates data-locality, thread-based programming and asynchronous communication. GPI (Global address space Programming Interface) [9][1] is a PGAS API for parallel applications running on clusters. The thin communication layer delivers the full performance of RDMA-enabled[2] networks directly to the application without interrupting the CPU. Fig. 1 depicts the architecture of GPI.

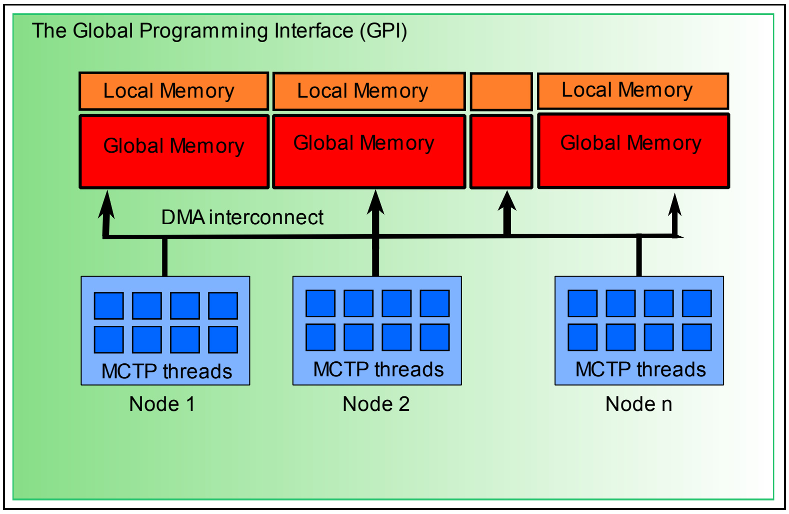


Fig. 1. GPI

The local memory is the internal memory available only to the node and allocated through typical allocators (e.g. malloc). This memory cannot be accessed by other nodes. The global memory is the partitioned global shared memory, available to other nodes, and where shared data should be placed. The DMA interconnect connects all nodes and is the underlying mechanism for most GPI operations. On each node, the Manycore Threading Package (MCTP) is used to take advantage of all cores present on the system. MCTP is a threading package based on thread pools that abstracts the native threads of the platform and a component of GPI.

The GPI core includes different functionalities but in the context of this work, the most important functionality is the read/write of global data.

Two operations exist to read and write from global memory independent of whether it is a local or remote location. One important point is that those operations are one-sided that is, only the peer that issues such operation takes part in it. This is different from a two-sided scheme (message passing) where the peer that sends (*sender*) has a corresponding peer (*receiver*) that needs to issue a receive operation. Moreover, this functionality is non-blocking and completely off-loaded to the interconnect, allowing the program to continue its execution

[1] GPI was previously known as Fraunhofer Virtual Machine (FVM).
[2] RDMA - Remote Direct Memory Access.

and hence take better advantage of CPU cycles. The data movement does not require any intermediate buffers and protocols to maintain those buffers. If the application needs to make sure the data was transferred (read or write), it needs to call a wait operation that blocks until the transfer is finished and asserting that the data is usable.

3 Adaptive Search

Local Search is based on the simple idea of "chasing" a solution by iteratively moving from one candidate (call this a "configuration") to one of its *neighbours*. The neighbourhood of a configuration is the set of configurations that can be obtained by applying a *move*. A *move* is a local change (hence the name Local Search).

The mechanism used to select a neighbour and thus the definition of what constitutes a neighbourhood is the main issue that differentiates between different local search methods. In general, it is problem dependent and is related to the definition of the *objective function*.

The Adaptive Search method [4] is one of many different local search methods and has proved to be very efficient in the types of problems where it was tested. It is a generic, domain-independent constraint-based local search method.

This meta-heuristic takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a single global cost function to optimize, such as for instance the number of violated constraints. The algorithm also uses a short-term adaptive memory in the spirit of Tabu Search [12] in order to prevent stagnation in local minima and loops. This method is generic, can be applied to a large class of constraints (e.g. linear and non-linear arithmetic constraints, symbolic constraints, etc) and naturally copes with over-constrained problems.

The input of the method is a problem in CSP format, that is, a set of variables with their (finite) domains of possible values and a set of constraints over these variables. For each constraint, an "error function" needs to be defined; it will give, for each tuple of variable values, an indication of how much the constraint is violated. For instance, the error function associated with an arithmetic constraint $|X - Y| < c$, for a given constant $c \geq 0$, could be $max(0, |X - Y| - c)$.

Adaptive Search relies on iterative repair, based on variable and constraint error information, seeking to reduce the error on the worst variable so far. The basic idea is to compute the error function for each constraint, then combine for each variable the errors of all constraints in which it appears, thereby projecting constraint errors onto the relevant variables. Finally, the variable with the highest error will be taken and its value will be modified. In this second step, the well known min-conflict heuristic is used to select the value in the variable domain which is the most promising, that is, the value for which the total error in the next configuration is least. In order to prevent being trapped in local minima, the Adaptive Search method also includes a short-term memory mechanism to store variables to avoid (variables can be marked Tabu and "frozen" for a number of iterations). It also integrates reset transitions to escape stagnation around local minima. A (partial) reset consists in assigning fresh random values to some variables (also randomly chosen). A reset is guided by the number of variables being marked Tabu. As in any local search method, it is also possible to restart from scratch when the number of iterations reaches a given limit.

4 Parallel Adaptive Search

When parallelizing an algorithm one aims at identifying hotspots and sources of parallelism. As with most meta-heuristics, in Adaptive Search these sources of parallelism are essentially: (1) the *inner loop* of the algorithm i.e., computing and combining the errors of variables and selecting the variable with highest error and (2) the *search space* of the problem.

The problem with exploiting the inner loop of the algorithm is its granularity: it is too fine-grained and the associated overhead might come at a too high cost.

The second main source of parallelism is the search space (domain) of the problem itself. Theoretically, this domain could be decomposed into several disjoint partitions, to be explored in parallel and without dependencies. However, in practice several issues arise with this: each partition is in general still too large for a sequential execution and, more importantly, the search space is not uniformly valid and the exploration should avoid areas that are known to lead to poor solutions. Moreover, it is hard and expensive to control and maintain the search conducted in the different partitions since a Local Search algorithm only has a local view of the search space. One example is the class of problems that have the best solutions clustered in a certain 'zone' of the search space. In this case, the algorithm should converge to that zone but in case of parallel execution avoid too much redundant work.

The Adaptive Search method has already been subject to some research on its parallel behaviour. Previous work on parallel implementations of the Adaptive Search algorithm have mostly focused on independent multiple-walks. Recal that independent multiple-walks are the simplest approach to parallel local search. A walk is carried out by each processor without any communication between them. Processors (search threads) start at a different solution and perform their own walk, intersecting or not, with walks from other processors. The same or different algorithms can be used to perform the walk, with the same or different parameters.

In [6], the authors present a parallel implementation of the Adaptive Search algorithm for the Cell/BE, a heterogenous multicore architecture. The system includes 16 processors (the SPEs[3]) where each one starts with a different random initial configuration. The PPE[4] acts as the master processor, waiting for the message of a found solution. For this number of processing units, the results were very promising, achieving linear speed-up for most problems.

[3] Synergistic Processing Element.
[4] Power Processor Element.

Further work with Parallel Adaptive Search continued to follow the same approach with no communication between workers, but more interestingly, concentrating on cluster systems with a larger number of cores.

In [2], the authors experiment and investigate the performance of a multiple independent-walk search on a system with up to 256 cores. The parallelization was done with MPI and involves the introduction of a "communication step" which tests if termination was detected (a solution was found) and terminates the execution properly. The results are relatively modest in terms of parallel efficiency, far for the ideal speed-up, which is in contrast with the results obtained at a smaller scale (on the Cell/BE, ie. with up to 16 cores). This points out the need for better alternative strategies in order to better exploit large-scale parallelism.

Since the independent multiple-walk approach still leaves space for improvement in terms of parallel efficiency and scalability for some problems, new ways to take full advantage of parallel systems must be found.

In [1], the authors experiment with more complex strategies, where processes exchange messages resembling branch-and-bound methods where the bound is exchanged between all participants. In their work, two alternatives are attempted: exchanging the cost of the current configuration of each process and the current cost plus the number of iterations needed to achieve that cost. Unfortunately, neither approach achieves better results than an independent multiple-walk.

5 Adaptive Search with GPI

Previous work with parallel Adaptive Search provides some groundwork to build upon and has shown that some benchmarks exhibit scalability problems when run on a large number of cores.

GPI appears to be, *à priori*, an interesting match to the problem of parallelizing Adaptive Search: local search methods work with local information, trying to progress and converge on solutions in a global search space, requiring little global information. However, as demonstrated by previous work, some problems exhibit low parallel efficiency and communication and cooperation becomes requirements to obtain good scalability. The communication with GPI is based on one-sided primitives that ought to benefit the local view on a global search space, as it allows threads to cooperate asynchronously. Moreover, communication is very efficient as GPI exploits the full performance of the interconnect with little or no CPU intervention. Hence, we continue to explore ways to further improve the parallelization of the Adaptive Search algorithm, exploiting GPI and its programming model, with the objective of getting some further benefits. But more importantly, to find mechanisms, concepts or limitations that are general.

In general, we can define the following objectives:

– further investigate and **understand the behaviour** of parallel Adaptive Search on different problems.

- investigate the **possibilites given by GPI** and devise more sophisticated mechanisms for the parallel execution of Adaptive Search, improving its performance
- identify the, possibly new, **problems** generated by the previous point.

The new parallel version of Adaptive Search based on GPI includes two variants which we name TDO (Termination Detection Only) and PoC (Propagation of Configuration).

The TDO variant implements the simple independent multiple-walk and serves mostly has our basis for comparison. First, with the existing MPI version, making sure that the implementation is correct and the performance is as expected. Second, to allow us to measure the improvement (if any) obtained with the more complex PoC variant. The PoC variant introduces more communication and sharing between working threads, by means of GPI primitives and its threaded model, but it is our expectation that this overhead will be offset by the performance gain.

The next sections present the two different variants in more detail.

5.1 Termination Detection Only

The variant with Termination Detection Only (TDO) is straightforward and implements the idea of *independent multiple-walks*: all available cores execute the sequential version of the Adaptive Search algorithm.

We name this variant "Termination Detection Only" because it amounts to a termination detection problem i.e., detecting the termination of a distributed computation. Termination Detection is in itself a subject of much research and several algorithms have been and continue to be proposed([7,10,15]).

In the case of the Parallel Adaptive Search method, we are interested in detecting termination as soon as one of the participating threads has found a solution: we want to get the first (earliest) solution. The implementation of this variant is simple as it only requires a triggering mechanism.

The GPI implementation follows a line similar to the previous work with MPI. Whenever a thread finds a solution, it triggers termination by writing to its peers that it has found a solution. Thus, the wall-clock time of the parallel execution is the time taken by this fastest thread.

Other threads must detect termination. This entails introducing a communication step in the internal loop of the Adaptive Search algorithm. This is required since there is no other way for a GPI instance to react on an remote event (i.e., termination) other than with communication. In this step, a check for termination is done on a particular memory address that is written on termination emission as described above. The communication step introduces some overhead that needs to be kept low, thus it is only executed every k iterations.

5.2 Propagation of Configuration

The experiments in previous work and with the TDO variant have found that the simple approach to parallelization, namely, the independent multiple-walk, is insufficient to obtain parallel efficiency on some problems especially when experimenting with a large number of cores. Moreover, exchanging simple information such as the cost leads to no improvement. This result goes to show that this is not a reliable metric, at least not by itself: it just says that cost C (better than the current cost) can be achieved but says nothing about when and how to reach it.

Hence, we aim at communicating more, and more meaningful information, introducing cooperation. By cooperation we mean mechanisms that allow threads to share information about their state and thus benefit from the collective search. Also, we would like to exploit the potential and benefits of GPI and its programming model (one-sided communication, no wait for communication, global access to data, threaded model, etc.) This can be achieved, for instance, by moving towards algorithms which resort to more communication than in previous cases.

One of the most powerful aspects of Local Search is its simplicity. Because of this, it is not obvious what could be considered as the *meaningful* information to be shared and communicated to other threads. One promising candidate which hasn't yet been tried is the whole current configuration.[5] The final configuration represents the solution when the algorithm stops.

The current implementation of the Adaptive Search method deals only with permutation problems and thus, a configuration is the permutation vector of the problems' variables.

Similarly to other approaches to parallelization which introduce cooperation, several important questions arise, namely:

1. Who does the communication?
2. When to do the communication?
3. How to do the communication?
4. What to communicate?

Our approach, which we call *Propagation of Configuration* (PoC), aims at answering these questions and giving a better understanding of how cooperation can help with increasing the scalability of Local Search in general and the Adaptive Search method in particular.

Who Does the Communication?

Note that on each node, there are as many threads as the number of availble cores. *Communication* is performed between nodes, by reading or writing the global memory of GPI. Hence, to answer this question we consider if, for each node, all or only a single thread actually performs communication with the other nodes.

There are potential advantages and disvantages with both options. If all threads perform communication, any shared resources must be protected by a mutual exclusion mechanism, which might suffer from high contention. Moreover,

[5] Because the term *solution* is sometimes misleading, we refer to the current solution as a *configuration*.

when all threads perform communication, a lot more pressure on the interconnect follows, increasing the parallel overhead and with possibly a lot of redundant communication happening (the same configuration being passed around several times). On the other hand, there will be a rapid progress towards the best promising neighbourhood, intensifying the search. Of course, this can be positive but can also become dangerous since most of threads might get trapped in a local minimum or poor quality neighbourhood. A good trade-off between intensification and diversification needs to be achieved.

If a single master thread communicates, the effects are potentially the opposite: less intensification but also less contention, less pressure on the interconnect and less redundant work.

Preliminary tests have made it clear that the best option is the one with a *single communicating thread* since it reduces the parallel overhead. Moreover, with GPI, all threads in a single node benefit immediately from the results obtained by the master thread without any messages exchange.

When to Do the Communication?

The first possible answer to this question is to follow the same strategy as with the Termination Detection Only variant: introduce a communication step and perform communication every k iterations. The value of k has a very significant impact on performance: with a low value (e.g., $k = 10$), a strong intensification of the search is achieved but with the danger that threads might give up too soon on a promising neighbourhood. With a high value of k, we avoid that danger but less intensification will be achieved since less information will be propagated.

The other option is to not interrupt the normal flow of the algorithm for communication, letting the search progress normally and independently until a local minimum is achieved. Only at this point the configuration is propagated and possibly used. One danger, however, is if threads do not hit local minima that often, the propagation of configuration will not progress and some threads might never see an up-to-date configuration. A solution to this problem is to still have communication every k iteration, where threads only use the propagated information when they are "in trouble" i.e., they hit a local minimum. However, this option increases the overhead by adding the extra communication step in some iterations.

In principle the second option might seem more promising as no disturbance is caused when the algorithm is progressing well. But the aforementioned danger that the propagation of configurations won't progress can have the consequence that there won't be any benefit from the communication scheme when compared to the simple TDO variant. Prelimary tests on a problem with low number of local minima (Magic Squares) confirmed this fact. Hence and based on this reasoning, we opted to have a communication step. Our PoC variant combines termination detection and the propagation of configurations in a single step that happens every k iterations and we focus the experimentation on finding an good value for k.

How to Do the Communication?

With this question, we consider a single alternative. Since we aim at large scale executions (hundreds to thousands of nodes), we need an efficient approach. Communication is done along a tree-based topology, where each node only communicates with its parent and children (if any). Currently, a binary tree is used but this can be parametrized at initialization. At each communication step, the propagation of the configuration is done either up (to parent) or down (to the children) the tree. This only happens if a configuration was propagated from the children (in case of the up direction) or from the parent (down direction). The propagation of the communication behaves then like a wave, up and down the tree, with possibly different configurations being propagated at different points of the tree and contributing to some diversification.

Communication is performed by using GPI one-sided primitives. A thread posts a write operation and returns immediately to work. The configuration to be propagated will be directly written to the memory of the remote node, asynchronously, without any acknowledgement and overlapped with the algorithm's computation. The remote node on the other hand, on its communication step, checks if a valid configuration was written to its memory, decides how to act on it and propagates its decision further.

We consider this single alternative since it gives us a good balance between intensification and diversification and because having a tree-based topology provides an efficient pattern to achieve communication scalability, with good locality. The final objective is to have a communication step with low overhead and here GPI provides us with mechanisms to do so.

What to Communicate?

The Adaptive Search method (as many other Local Search methods) is very simple and includes very few elements that can be communicated.

The proposed option has been already mentioned: to *communicate a full configuration*. To this, we only add the cost of the configuration as it is the metric to evaluate the configuration, and we need only compute it once.

Still, the question remains of which configuration to communicate. In our design the best configuration (with better cost) is communicated. At a communication step, a thread decides whether to propagate its own current configuration or the propagated configuration(s) it received from its neighbour(s).

Communicating configurations is advantageous because configurations implicitly contain more information about the global state of the search: as the best configurations are being propagated, threads that are currently on poorer neighbourhoods might benefit from moving to the best ones. With the stochastic behaviour of Adaptive Search and enough diversification, the whole search procedure can be performed on the best neighbourhoods and, hopefully, converge faster onto good solutions.

6 Experimental Results

In this section we present the obtained results using a few benchmark problems.

– **costas-array**: the Costas Array problem [5],
– **all-interval**: the All Interval Series problem (`prob007` in CSPLib [8]),
– **magic-square**: the Magic Square problem (`prob019` in CSPLib).

The experiments were conducted on a cluster system where each node includes a dual Intel Xeon 5148LV "Woodcrest" (i.e., 4 CPUs per node) with 8 GB of RAM. The full system is composed of 620 cores connected with Infiniband (DDR). We performed our experiments on the system using up to 256 cores on some problems and 512 cores on others. The difference is due to the fact that the system is heavily in use and it is hard to get access to the full cluster.

Note that Adaptive Search, as many other Local Search methods, has a stochastic behaviour to achieve diversity on the search. To benchmark such behaviour, several executions must be done and averaged. In our experiments we ran each problem 100 times in order to obtain meaningful results.

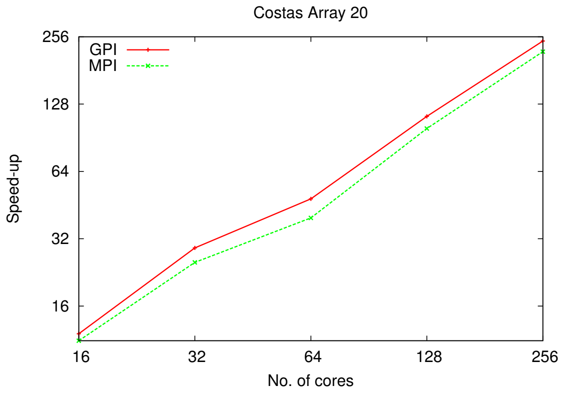


Fig. 2. CAP(20) on 256 cores (64 nodes)

We compare both GPI variants (TDO and PoC) with the MPI implementation, as a basis for comparison. Figure 2 depicts the obtained results for the Costas Array problem (CAP) with n=20.

As already observed in [3], the CAP shows an almost optimal scalability using an independent multiple -walk with no cooperation. We can observe that our implementation obtains similar, although slight better, results. This is the expected result since both approaches (TDO and MPI) are equivalent: communication is only used for detecting termination. Nevertheless, it is a confirmation that our implementation performs as expected.

Although we aspired at obtaining even better results with the PoC variant (possibly super linear) for this problem, our experiments showed that this variant performs much worse than the simple TDO variant and thus we only present the speedup obtained with GPI using the TDO variant.

Figure 3 depicts the obtained results for the Magic Square problem up to 512 cores. For this problem we present the speedup obtained with the TDO and PoC variants and compare it with the MPI version. The GPI TDO variant presents again, as expected, results similar to the MPI version.

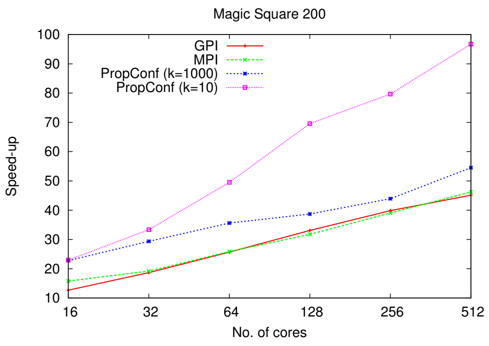


Fig. 3. MS(200) on 512 cores (128 nodes)

The Magic Square problem is one of the problems that results in disappointing scalability when using the simple independent multiple-walk and therefore a major target for improvement with more sophisticated approaches. Indeed, for this problem, our PoC variant improves the performance and scales better as we increase the number of cores used.

We wanted to answer the question of when to do communication: as we mentioned, in our preliminary experiments it turned out that the best approach is to have a communication step every k iterations where the value of k is decisive. Surprinsingly, for this problem, a lower value of k (k=10 in contrast to k=1000) improves scalability by a factor of 2, achieving a speedup of 97 with 512 cores. Still a low parallel efficiency but a very significant improvement over the other options and variants.

The results we obtained for the last problem, the All Interval series (n=400), are shown in Figure 4.

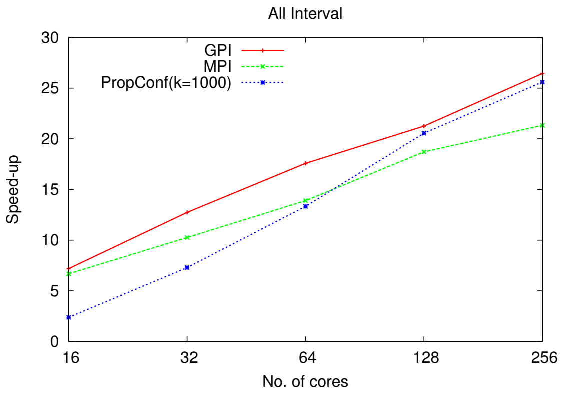


Fig. 4. AI(400) on 256 cores (64 nodes)

The All Interval Series benchmark is also one of the problems where good scalability was hard to reach when using a large number of cores. In Figure 4, one may observe this fact where both the MPI and GPI TDO versions reach a modest speedup factor of 20 and 25, respectively (with 256 cores). Our PoC variant however, performs much worse than the TDO variant at a low number of cores but it improves as we increase the number of cores, hinting that this variant can be of advantage if we increase the number of cores and the problem size. In Fig. 4 we only depict the obtained results for the PoC variant with k = 1000 since, for this benchmark, it is the best value. In contrast to the Magic Squares benchmark, a lower value of k results in a much worse performance.

7 Discussion

The experimental results presented large differences in how the different problems benefit from parallelism and the implemented variants. One of our main objectives is to investigate and understand why this happens.

In order to be able to draw some conclusions on our experiments, it is important to characterize the chosen problems from different perspectives. We do so, resorting to different information such as the number of iterations and local minima. This characterization will give us a basis to better understand the problems at hand and ultimately explain our results.

Table 1 presents the obtained values for acquired information when running some instances of the previously presented problems. This information is the following:

Problem The problem instance.
Iterations The number of iterations required to find a solution.
Local Minima The number of local minima found.
Resets The number of partial resets performed (not full restart).
Same var / Iteration The number of times there was more than one candidate variable (highest error value) to be chosen from.

This information allows us to better understand how does the Adaptive Search algorithm progress towards a solution, the neighbourhood structure and extract further information (e.g., number of local minima *per* iteration).

Table 1. Information collected for different problems instances

Problem	Iterations	Local Minima	Resets	Same var/ Iteration
Magic Square 200	413900	25864	3	23.36
Costas 18	389932	204024	204024	1.00
Costas 19	3364807	1714299	1714299	0.99
All Interval 200	11229	495	495	5.97
All Interval 400	41122	1422	1422	9.19

From Table 1 we can see that the different problems exhibit a significantly different behaviour. *Magic Square* performs a low number of partial resets when compared to the total number of iterations or to the number of identified local minima. On the other hand, it is the problem where the number of candidate variables per iterations (Same var/Iteration) is highest, meaning that at each iteration there are several possible moves towards the next configuration.

The *Costas Array* problem exhibits a completely different behaviour. In this case, the number of identified local minima is very large (almost every second iteration finds a local minimum) and the number of partial resets is also very high, coincident with the number of local minima i.e., at each local minimum found, a partial reset is performed. Also the number of possible moves at each iteration is close to 1.

All Interval is yet another kind of problem. Here, the number of resets is as with the CAP equal to the number of local minima but these happen much less often. The number of possible variable choices or moves is higher than 1, meaning that some diversification could be achieved.

If we relate this characterization of problems with the obtained experimental results, some conclusions can be drawn in order to better understand the parallelization behaviour of this algorithm or, more concretely, how much can it benefit from a communication scheme such as the one we designed.

We argue that one critical aspect is the density of the *neighbourhood* of a configuration or the set of possible moves, which define transitions between configurations. Since we are propagating configurations we can look at our problems at hand according to this aspect. If a problem has a dense neighbourhood or, in other words, the set of possible moves at each transition is (much) larger than one, each of these moves can be explored in parallel. Thus, when a promising configuration is propagated and several moves are possible and explored in parallel, the probability that one of these moves leads to a faster path towards an optimal solution increases.

Another important aspect is the number of *local minima and resets* and how both relate. A problem that finds a large number of local minima before encountering an optimal solution benefits less from processing a configuration which seems promising. This configuration is heuristically promising but in reality this information is less meaningful than it should. Similarly, a problem with a high number of partial resets suffers from the same problem.

Looking back at our experimental results with the different problems, we can better understand a) the difference in scalability and b) the improvement factor brought by the PoC variant to some problems.

In the Magic Square problem, each configuration has a dense neighbourhood and benefits from the parallel exploration of different moves. Thus, the PoC variant improves the performance and scalability of the algorithm. When a working thread adopts a propagated configuration, it will define its own path from that configuration and differently from one other thread that receives that same promising configuration. Moreover, this problem has a low number of local minima and resets meaning that paths from one (initial) configuration towards an optimal solution are a series of transitions from neighbour configurations.

The Costas Array Problem exhibits optimal scalability with the independent multiple-walk MPI version or with our TDO variant and this is already *per se* satisfactory. On the other hand, it performs worse with the PoC variant: propagating a configuration is only a source of parallel overhead and will limit the search allowing less diversification. A propagated configuration will allow, on average, a single move and two threads taking the same configuration results in redundant work. This is also probably unfruitful since the CAP is one of the problems with a high number of local minima and resets. This also explains the good scalability using the TDO variant, where increasing the number of cores allows covering more of the total search space together with the fact that solutions for this problem are well spread over it.

Finally, the All Interval Series problem shows a mixed behaviour. Similarly to the CAP, the larger number of local minima found and same number of resets point to the same situation: there is less benefit from taking a propagated configuration since its meaningfulness is low. The PoC variant only introduces unnecessary overhead and this could explain the much worse performance at a lower number of cores. On the other hand, and similarly to the Magic Square benchmark, there is more than one possible move, on average i.e., some diversification can be achieved. With a large enough number of cores, the parallel overhead can be amortized by the gain obtained with this diversification. This could be the reason for the steeper curve for the PoC variant on Fig. 4. Of course, with further experiments we will be able to understand this better.

In summary, problems that follow a trajectory with a single possible move won't benefit from a communication scheme that propagates the best current configuration(s). Also, if a large number of local minima is found and partial resets are required in the same number, the expectation for improvement in performance is rather low. On the contrary, problems where configurations have a denser neighbourhood benefit from a cooperation scheme such as the PoC variant where the full configuration is communicated and improvements in performance are expected.

8 Conclusion

In this paper we presented our work on the parallel implementation of the Adaptive Search method using an alternative programming model. GPI is an API designed for high-performance and scalable parallel applications. We aimed at investigating and understanding the behaviour of Adaptive Search in a parallel setting, focusing on different problems particularly those that, in previous work, showed scalability problems when targeting a large number of cores. GPI and its programming model allowed us to design a new communication and parallelization scheme which in our experimental evalution allowed a gain of a factor of 2 in terms of speedup for some problems. More importantly, it provided deeper insight and understanding on the parallelization of Local Search methods given different problems with disparate characteristics such as the density of a configuration neighbourhood, the number of local minima and partial resets.

We point out that GPI performs well and allows us to adopt more communication -intensive schemes, which supports the claim that solving local search problems is a good use case for GPI.

In the future, we intend to examine our design and conclusions with other larger problems and experiment with more sophisticated parallelization schemes. One possible direction is, instead of using promising information (configurations, cost, statistics) directly, to act on its complement, avoiding redundant work and thereby cover as much as possible from the search space since this is the main source of parallelism. Another direction is to revisit the modeling of each problem knowing that it will be executed in parallel; this is relevant as the current models are designed and optimized for sequential execution. Models designed with parallelism in mind, even if less efficient in serial execution, will benefit at scale as more cores are used in the solving process.

One of our potential final goals is the design of a new Local Search algorithm based on Adaptive Search and more amenable to parallelization, building upon the experiences presented in this paper.

Ultimately, the work described herein will be integrated with MaCS, a GPI port of the PaCCS hierarchical distributed constraint solving system [13], providing additional insight on how to reach good parallel performance on CSPs.

Acknowledgements. This work was partly supported by Fundação para a Ciência e Tecnologia under grant PTDC/EIA-EIA/100897/2008 (HORUS).

References

1. Caniou, Y., Codognet, P.: Communication in parallel algorithms for constraint-based local search. In: IPDPS Workshops, pp. 1961–1970 (2011)
2. Caniou, Y., Codognet, P., Diaz, D., Abreu, S.: Experiments in parallel constraint-based local search. In: Merz, P., Hao, J.-K. (eds.) EvoCOP 2011. LNCS, vol. 6622, pp. 96–107. Springer, Heidelberg (2011)
3. Caniou, Y., Diaz, D., Richoux, F., Codognet, P., Abreu, S.: Performance analysis of parallel constraint-based local search. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, New Orleans, Louisiana, USA, pp. 337–338. ACM, New York (2012), `http://doi.acm.org/10.1145/2145816.2145883`, doi:10.1145/2145816.2145883
4. Codognet, P., Diaz, D.: Yet another local search method for constraint solving. Stochastic Algorithms: Foundations and Applications, 342–344 (2001)
5. Costas, J.: A study of detection waveforms having nearly ideal range-doppler ambiguity properties. Proceedings of the IEEE 72(8), 996–1009 (1984)
6. Diaz, D., Abreu, S., Codognet, P.: Targeting the cell broadband engine for constraint-based local search. Concurrency and Computation: Practice and Experience 24(6), 647–660 (2012)
7. Dijkstra, E.W., Feijen, W.H.J., van Gasteren, A.J.M.: Derivation of a termination detection algorithm for distributed computations. Inf. Process. Lett. 16(5), 217–219 (1983)
8. Gent, I.P., Walsh, T.: Csplib: A benchmark library for constraints. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 480–481. Springer, Heidelberg (1999), `http://www.csplib.org`
9. Machado, R., Lojewski, C.: The fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. Computer Science-Research and Development 23(3), 125–132 (2009)
10. Mattern, F.: Algorithms for distributed termination detection. Distributed Computing 2, 161–175 (1987), doi:10.1007/BF01782776

11. MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2 (September 4, 2009) (December 2009), http://www.mpi-forum.org
12. Pardalos, P.M., Pitsoulis, L., Mavridou, T., Resende, M.G.C.: Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing, tabu search and grasp. In: Ferreira, A., Rolim, J. (eds.) IRREGULAR 1995. LNCS, vol. 980, pp. 317–331. Springer, Heidelberg (1995)
13. Pedro, V., Machado, R., Abreu, S.: A Parallel and Distributed Framework for Constraint Solving. In: Proceedings of the 1st Workshop on Parallel Methods for Constraint Solving, PCMS 2011 (2011)
14. Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming, vol. 2. Elsevier Science (2006)
15. Saraswat, V.A., Kambadur, P., Kodali, S.B., Grove, D., Krishnamoorthy, S.: Lifeline-based global load balancing. In: PPOPP, pp. 201–212 (2011)

Author Index