LNCS 5979

Santiago Escobar (Ed.)

Functional and Constraint Logic Programming

18th International Workshop, WFLP 2009
Brasilia, Brazil, June 2009
Revised Selected Papers

Springer

Santiago Escobar (Ed.)

Functional and Constraint Logic Programming

18th International Workshop, WFLP 2009
Brasilia, Brazil, June 28, 2009
Revised Selected Papers

Springer

Volume Editor

Santiago Escobar
Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación
Camino de vera, s/n, 46022 Valencia, Spain
E-mail: sescobar@dsic.upv.es

Library of Congress Control Number: 2010923100

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

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

ISSN	0302-9743
ISBN-10	3-642-11998-0 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-11998-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains a selection of the papers presented at the 18th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2009), held on June 28, 2009 in Brasília, Brazil. Previous WFLP meetings were held in Siena (2008), Paris (2007), Madrid (2006), Tallinn (2005), Aachen (2004), Valencia (2003), Grado (2002), Kiel (2001), Benicassim (2000), Grenoble (1999), Bad Honnef (1998), Schwarzenberg (1997, 1995, and 1994), Marburg (1996), Rattenberg (1993), and Karlsruhe (1992).

The aim of the WFLP series is to bring together researchers interested in functional programming, (constraint) logic programming, as well as the integration of the two paradigms. It promotes the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications, and combinations of high-level, declarative programming languages and related areas.

WFLP 2009 solicited papers in all areas of functional and (constraint) logic programming, including but not limited to:

- Foundations: formal semantics, rewriting and narrowing, non-monotonic reasoning, dynamics, and type theory.
- Language Design: modules and type systems, multi-paradigm languages, concurrency and distribution, and objects.
- Implementation: abstract machines, parallelism, compile-time and run-time optimizations, and interfacing with external languages.
- Transformation and Analysis: abstract interpretation, specialization, partial evaluation, program transformation, and meta-programming.
- Software Engineering: design patterns, specification, verification and validation, debugging, and test generation.
- Integration of Paradigms: integration of declarative programming with other paradigms such as imperative, object-oriented, concurrent, and real-time programming.
- Applications: declarative programming in education and industry, domain-specific languages, visual/graphical user interfaces, embedded systems, WWW applications, knowledge representation and machine learning, deductive databases, advanced programming environments and tools.

The WFLP 2009 workshop was part of the Federated Conference on Rewriting, Deduction, and Programming (RDP 2009), which grouped together different events such as the 20th International Conference on Rewriting Techniques and Applications (RTA 2009), the 9th International Conference on Typed Lambda Calculi and Applications (TLCA 2009), the 4th Workshop on Logical and Semantic Frameworks, with Applications (LFSA 2009), the 10th International Workshop on Rule-Based Programming (RULE 2009), and the 9th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2009).

There were 14 original contributions to the workshop, the Program Committee selected nine papers for publication, and revised versions of these selected papers are included in this volume. Each contribution was reviewed by at least three Program Committee members. This volume also includes two invited contributions by Claude Kirchner from the Centre de Recherche INRIA Bordeaux - Sud-Ouest, France, and Roberto Ierusalimschy from the Departamento de Informática, PUC-Rio, Brazil. I would like to thank them for having accepted our invitation for both the scientific program and this volume. I am also grateful to Andrei Voronkov for his extremely useful EasyChair system for automation of conference chairing.

I would also like to thank all the members of the Program Committee and all the referees for their careful work in the review and selection process. Many thanks to all authors who submitted papers and to all conference participants. I gratefully acknowledge the *Departamento de Sistemas Informáticos y Computación* of the *Universidad Politécnica de Valencia*, who supported this event. Finally, I express our gratitude to all members of the Local Organization of the Federated Conference on Rewriting, Deduction, and Programming (RDP 2009), whose work made the workshop possible.

December 2009 Santiago Escobar

Organization

Program Chair

Santiago Escobar
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de vera, s/n
E-46022 Valencia, Spain
`sescobar@dsic.upv.es`

Program Committee

María Alpuente	Universidad Politécnica de Valencia, Spain
Sergio Antoy	Portland State University, USA
Christiano Braga	Universidade Federal Fluminense, Brazil
Rafael Caballero	Universidad Complutense de Madrid, Spain
David Déharbe	Universidade Federal do Rio Grande do Norte, Brazil
Rachid Echahed	CNRS, Laboratoire LIG, France
Moreno Falaschi	Università di Siena, Italy
Michael Hanus	Christian-Albrechts-Universität zu Kiel, Germany
Frank Huch	Christian-Albrechts-Universität zu Kiel, Germany
Tetsuo Ida	University of Tsukuba, Japan
Wolfgang Lux	Westfalische Wilhelms-Universität Münster, Germany
Mircea Marin	University of Tsukuba, Japan
Camilo Rueda	Universidad Javeriana-Cali, Colombia
Jaime Sánchez-Hernández	Universidad Complutense de Madrid, Spain
Anderson Santana de Oliveira	Universidade Federal do Rio Grande do Norte, Brazil

Additional Reviewers

Hassan Aït-Kaci
Gloria Alvarez
Demis Ballis
Bernd Braßel
Linda Brodo
Iliano Cervesato
Yukiyoshi Kameyama
Temur Kutsia
Michael Maher
Miguel Palomino
Cody Roux
Albert Rubio
Clara Segura
Peter Sestoft
Nikhil Swamy
Thierry Boy de la Tour
Rafael del Vado Vrseda
Toshiyuki Yamada
Hans Zantema

Table of Contents

Programming with Multiple Paradigms in Lua

Roberto Ierusalimschy

PUC-Rio, Rio de Janeiro, Brazil
roberto@inf.puc-rio.br

Abstract. Lua is a scripting language used in many industrial applications, with an emphasis on embedded systems and games. Two key points in the design of the language that led to its widely adoption are flexibility and small size. To achieve these two conflicting goals, the design emphasizes the use of few but powerful mechanisms, such as first-class functions, associative arrays, coroutines, and reflexive capabilities. As a consequence of this design, although Lua is primarily a procedural language, it is frequently used in several different programming paradigms, such as functional, object-oriented, goal-oriented, and concurrent programming, and also for data description.

In this paper we discuss what mechanisms Lua features to achieve its flexibility and how programmers use them for different paradigms.

1 Introduction

Lua is an embeddable scripting language used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems and games. It is embedded in devices ranging from cameras (Canon) to keyboards (Logitech G15) to network security appliances (Cisco ASA). In 2003 it was voted the most popular language for scripting games by a poll on the site Gamedev[1]. In 2006 it was called a "de facto standard for game scripting" [1]. Lua is also part of the Brazilian standard middleware for digital TV [2].

Like many other languages, Lua strives to be a flexible language. However, Lua also strives to be a small language, both in terms of its specification and its implementation. This is an important feature for an embeddable language that frequently is used in devices with limited hardware resources [3]. To achieve these two conflicting goals, the design of Lua has always been economical about new features. It emphasizes the use of few but powerful mechanisms, such as first-class functions, associative arrays, coroutines, and reflexive capabilities [4,5].

Lua has several similarities with Scheme, despite a very different syntax. (Lua adopts a conventional syntax instead of Lisp's S-expressions.) Both languages are dynamically typed. As in Scheme, all functions in Lua are anonymous first-class values with lexical scoping; a "function name" is just the name of a regular variable that refers to that function. As in Scheme, Lua does proper tail calls. Lua also offers a single unifying data-structure mechanism.

[1] http://www.gamedev.net/gdpolls/viewpoll.asp?ID=163

However, to keep the language and its implementation small, Lua is more pragmatic than Scheme. Its main data structure is the *table*, or associative arrays, instead of lists. (The former seems a better fit for procedural programming, while the latter seems better for functional programming.) Instead of a hierarchy of numerical types —real, rational, integer— in Lua all numbers are double floating-point values. (With this representation, Lua transfers all the burden of numeric specification and implementation to the underlying system.) Instead of full continuations, Lua offers one-shot continuations in the form of *stackfull* coroutines [6]. (Efficient implementations of coroutines are much simpler than efficient implementations of full continuations and most uses of continuations can be done with coroutines/one-shot continuations.)

As an authentic scripting language, a design goal of Lua is to offer strong support for dual-language programming [7]. The API with C is a key ingredient of the language. To easy the integration between the scripting language and the host language, Lua is amenable to different kinds of programming: event-driven, object oriented, etc. Moreover, to better explore the flexibility offered by the language, Lua programmers frequently use several paradigms, such as functional, object-oriented, goal-oriented, and concurrent programming, and also data description.

In this paper we discuss what mechanisms Lua features to achieve its flexibility and how programmers use them for different paradigms. The rest of the paper is organized around different paradigms. The next section describes the uses of Lua for data description. Section 3 discusses Lua support for functional programming. Section 4 discusses object-oriented programming in Lua. Section 5 shows how we can use coroutines to implement goal-oriented programming, where a *goal* is either a primitive goal or a disjunction of alternative goals. Section 6 discusses two ways to implement concurrency in Lua: *collaborative multithreading*, using coroutines, and *Lua processes*, using multiple independent states. Finally, Section 7 draws some conclusions.

2 Data Description

Lua was born from a data-description language, called SOL [8], a language somewhat similar to XML in intent. Lua inherited from SOL the support for data description, but integrated that support into its procedural semantics.

SOL was somewhat inspired by BibTeX, a tool for creating and formating lists of bibliographic references [9]. A main difference between SOL and BibTeX was that SOL had the ability to name and nest declarations. Figure 1 shows a typical fragment of SOL code, slightly adapted to meet the current syntax of Lua. SOL acted like an XML DOM reader, reading the data file and building an internal tree representing that data; an application then could use an API to traverse that tree.

Lua mostly kept the original SOL syntax, with small changes. The semantics, however, was very different. In Lua, the code in Figure 1 is an imperative program. The syntax `{first = "Daniel", ...}` is a *constructor*: it builds a table,

```
dan = name{first = "Daniel", last = "Friedman"}

mitch = name{last = "Wand",
             first = "Mitchell",
             middle = "P."}

chris = name{first = "Christopher", last = "Haynes"}

book{"essentials",
  author = {dan, mitch, chris},
  title = "Essentials of Programming Languages",
  edition = 2,
  year = 2001,
  publisher = "The MIT Press"
}
```

Fig. 1. Data description with SOL/Lua

or associative array, with the given keys and values. The syntax `name{...}` is sugar for `name({...})`, that is, it builds a table and calls function `name` with that table as the sole argument. The syntax `{dan,mitch,chris}` again builds a table, but this time with implicit integer keys 1, 2, and 3, therefore representing a list. A program loading such a file should previously define functions `name` and `book` with appropriate behavior. For instance, function `book` could add the table to some internal list for later treatment.

Several applications use Lua for data description. Games frequently use Lua to describe characters and scenes. HiQLab, a tool for simulating high frequency resonators, uses Lua to describe finite-element meshes [10]. GUPPY uses Lua to describe sequence annotation data from genome databases [11]. Some descriptions comprise thousands of elements running for a few million lines of code. The user sees these files as data files, but Lua sees them as regular code. These huge "programs" pose a heavy load on the Lua precompiler. To handle such files efficiently, and also for simplicity, Lua uses a one-pass compiler with no intermediate representations. As we will see in the next section, this requirement puts a burden on other aspects of the implementation.

3 Functional Programming

Lua offers first-class functions with lexical scoping. For instance, the following code is valid Lua code:

```
(function (a,b) print(a+b) end)(10, 20)
```

It creates an anonymous function that prints the sum of its two parameters and applies that function to arguments 10 and 20.

All functions in Lua are anonymous dynamic values, created at run time. Lua offers a quite conventional syntax for creating functions, like in the following definition of a factorial function:

```
function fact (n)
  if n <= 1 then return 1
  else return n * fact(n - 1)
  end
end
```

However, this syntax is simply sugar for an assignment:

```
fact = function (n)
         ...
       end
```

This is quite similar to a `define` in Scheme [12].

Lua does not offer a *letrec* primitive. Instead, it relies on assignment to close a recursive reference. For instance, a strict recursive fixed-point operator can be defined like this:

```
local Y
Y = function (f)
      return function (x)
               return f(Y(f))(x)
             end
    end
```

Or, using some syntactic sugar, like this:

```
local function Y (f)
      return function (x)
               return f(Y(f))(x)
             end
    end
```

This second fragment expands to the first one. In both cases, the `Y` in the function body is bounded to the previously declared local variable.

Of course, we can also define a strict non-recursive fixed-point combinator in Lua:

```
Y = function (le)
      local a = function (f)
        return le(function (x) return f(f)(x) end)
      end
      return a(a)
    end
```

Despite being a procedural language, Lua frequently uses function values. Several functions in the standard Lua library are higher-order. For instance, the `sort` function accepts a comparison function as argument. In its pattern-matching functions, text substitution accepts a *replacement* function that receives the original text matching the pattern and returns its replacement. The standard library also offers some *traversal functions*, which receive a function to be applied to every element of a collection.

Most programming techniques for strict functional programming also work without modifications in Lua. As an example, LuaSocket, the standard library for network connection in Lua, uses functions to allow easy composition of different functionalities when reading from and writing to sockets [13].

Lua also features proper tail calls. Again, although this is a feature from the functional world, it has several interesting uses in procedural programs. For instance, it is used in a standard technique for implementing state machines [4]. In these implementations, each state is represented by a function, and tail calls transfer the program from one state to another.

Closures

The standard technique for implementing strict first-class functions with lexical scoping is with the use of closures. Most implementations of closures neglect assignment. Pure functional languages do not have assignment. In ML assignable cells have no names, so the problem of assignment to lexical-scoped variables does not arise. Since Rabbit [14], most Scheme compilers do *assignment conversions* [15], that is, they implement assignable variables as ML cells on the correct ground that they are not used often.

None of those implementations fit Lua, a procedural language where assignment is the norm. Moreover, as we already mentioned, Lua has an added requirement that its compiler must be fast, to handle huge data-description "programs", and small. So, Lua uses a simple one-pass compiler with no intermediate representations which cannot perform even escape analysis.

Due to these technical restrictions, previous versions of Lua offered a restricted form of lexical scoping. In that restricted form, a nested function could access the value of an outer variable, but could not assign to such variable. Moreover, the value accessed was frozen when the closure was created. Lua version 5, released in 2003, came with a novel technique for implementing closures that satisfies the following requirements [16]:

- It does not impact the performance of code that does not use non-local variables.
- It has an acceptable performance for imperative programs, where side effects (assignment) are the norm.
- It correctly handles *sharing*, where more than one closure modifies a non-local variable.
- It is compatible with the standard execution model for procedural languages, where variables live in activation records allocated in an array-based stack.
- It is amenable to a one-pass compiler that generates code on the fly, without intermediate representations.

4 Object-Oriented Programming

Lua has only one data-structure mechanism, the *table*. Tables are first-class, dynamically created associative arrays.

Tables plus first-class functions already give Lua partial support for objects. An object may be represented by a table: instance variables are regular table fields and methods are table fields containing functions. In particular, tables have identity. That is, a table is different from other tables even if they have the same contents, but it is equal to itself even if it changes its contents over time.

One missing ingredient in the mix of tables with first-class functions is how to connect method calls with their respective objects. If `obj` is a table with a *method* `foo` and we call `obj.foo()`, `foo` will have no reference to `obj`. We could solve this problem by making `foo` a closure with an internal reference to `obj`, but that is expensive, as each object would need its own closure for each of its methods.

A better mechanism would be to pass the receiver as a hidden argument to the method, as most object-oriented languages do. Lua supports this mechanism with a dedicated syntactic sugar, the *colon operator*: the syntax `orb:foo()` is sugar for `orb.foo(orb)`, so that the receiver is passed as an extra argument to the method. There is a similar sugar for method definitions. The syntax

```
function obj:foo (...) ... end
```

is sugar for

```
obj.foo = function (self, ...) ... end
```

That is, the colon adds an extra parameter to the function, with the fixed name `self`. The function body then may access instance variables as regular fields of table `self`.

To implement classes and inheritance, Lua uses delegation [17,18]. Delegation in Lua is very simple and is not directly connected with object-oriented programming; it is a concept that applies to any table. Any table may have a designated "parent" table. Whenever Lua fails to find a field in a table, it tries to find that field in the parent table. In other words, Lua delegates field accesses instead of method calls.

Let us see how this works. Let us assume an object `obj` and a call `obj:foo()`. This call actually means `obj.foo(obj)`, so Lua first looks for the key `foo` in table `obj`. If `obj` has such field, the call proceeds normally. Otherwise, Lua looks for that key in the parent of `obj`. Once it found a value for that key, Lua calls the value (which should be a function) with the original object `obj` as the first argument, so that `obj` becomes the value of the parameter `self` inside the method's body.

With delegation, a class is simply an object that keeps methods to be used by its instances. A class object typically has *constructor* methods too, which are used by the class itself. A constructor method creates a new table and makes it delegates its accesses to the class, so that any class method works over the new object.

If the parent object has a parent, the query for a method may trigger another query in the parent's parent, and so on. Therefore, we may use the same delegation mechanism to implement inheritance. In this setting, an object representing a (sub)class delegates accesses to unknown methods to another object representing its superclass.

For more advanced uses, a program may set a function as the parent of a table. In that case, whenever Lua cannot find a key in the table it calls the parent function to do the query. This mechanism allows several useful patterns, such as multiple inheritance and inter-language inheritance (where a Lua object may delegate to a C object, for instance).

5 Goal-Oriented Programming

Goal-oriented programming involves solving a *goal* that is either a primitive goal or a disjunction of alternative goals. These alternative goals may be, in turn, conjunctions of subgoals that must be satisfied in succession, each of them giving a partial outcome to the final result. Two typical examples of goal-oriented programming are text pattern matching [19] and Prolog-like queries [20].

In pattern-matching problems, the primitive goal is the matching of string literals, disjunctions are alternative patterns, and conjunctions represent sequences. In Prolog, the unification process is an example of a primitive goal, a relation constitutes a disjunction, and rules are conjunctions. In those contexts, a problem solver uses a backtracking mechanism that successively tries each alternative until it finds an adequate result.

A main problem when implementing problem solvers in conventional programming languages is that it is difficult to find an architecture that keeps the *principle of compositionality*. Following this principle, a piece of code that solves a problem should be some composition of the pieces that solve the subproblems. Because each subproblem may have more than one possible solution, an adequate architecture should provide an efficient way for each subproblem to produce its solutions one by one, by demand.

Lazy functional languages provide an interesting architecture for problem solving: the piece of code that solves a problem simply returns a list of all possible solutions [21]. Laziness ensures that the code actually produces only the solutions needed to find a global solution for the entire problem.

In Lua, we can use coroutines [22] for the task. A well-known model for Prolog-style backtracking is the *two-continuation model* [23,24]. Although this model requires multi-shot continuations, it is not difficult to adapt it to coroutines that are equivalent to one-shot continuations [25,6]. The important point is that the coroutine model keeps the principle of compositionality for the resulting system, as we will see in the following example.

Figure 2 shows a simple implementation of a pattern-matching library, taken from [6]. Each pattern is represented by a function that receives the subject plus

```
-- matching any character (primitive goal)
function any (S, pos)
  if pos < string.len(S) then coroutine.yield(pos + 1) end
end

-- matching a string literal (primitive goal)
function lit (str)
  local len = string.len(str)
  return function (S, pos)
           if string.sub(S, pos, pos+len-1) == str then
             coroutine.yield(pos+len)
           end
         end
end

-- alternative patterns (disjunction)
function alt (patt1, patt2)
  return function (S, pos)
           patt1(S, pos); patt2(S, pos)
         end
end

-- sequence of sub-patterns (conjunction)
function seq (patt1, patt2)
  return function (S, pos)
           local btpoint = coroutine.wrap(function() patt1(S, pos) end)
           for npos in btpoint do patt2(S, npos) end
       end
end
```

Fig. 2. A simple pattern-matching library

the current position and yields each possible final position for that match. More specifically, the code for a pattern yields all values j such that $\mathrm{sub}(s, i, j-1)$ (that is, the substring of s from i to $j-1$) matches the pattern.

Function `any` is a primitive pattern that matches any character. Function `lit` builds a primitive pattern that matches a literal string. Its resulting function only checks whether the substring from the subject starting at the current position is equal to the literal pattern; if so it yields the next position, otherwise it ends without yielding any option.

Function `alt` builds an alternative of two patterns: it simply calls the first one and then the second one. Each subpattern will yield its possible matchings.

Finally, function `seq` builds a sequence of two patterns. It runs the first one inside a new coroutine to collect its possible results and runs the second pattern for each of these results.

The next fragment shows a simple use:

```
-- subject
s = "abaabcda"
 -- pattern:   (.|ab)..
p = seq(alt(any, lit("ab")), seq(any, any))
seq(p, print)(s, 1)
-- results
--> abaabcda  4
--> abaabcda  5
```

It “sequences” the pattern with the `print` function, which prints its arguments (the subject plus the current position after matching `p`), and then calls the resulting pattern with the subject and the initial position (1).

6 Concurrent Programming

Traditional multithreading, which combines preemption and shared memory, is difficult to program and prone to errors [26]. Lua avoids the problems of traditional multithreading by cutting either preemption or shared memory.

To achieve multithreading without preemption, Lua uses coroutines. A *stackful* coroutine [6] is essentially a thread; it is easy to write a simple scheduler with a few lines of code to complete the system. The book *Programming in Lua* [4] shows an implementation for a primitive multithreading system with less than 50 lines of Lua code.

This combination of coroutines with a scheduler results in collaborative multithreading, where each thread should explicitly yield periodically. This kind of concurrency seems particularly apt for simulation systems and games.[2]

Coroutines offer a very light form of concurrency. In a regular PC, a program may create tens of thousands of coroutines without draining system resources. Resuming or yielding a coroutine is slightly more expensive than a function call. Games, for instance, may easily dedicate a coroutine for each relevant object in the game.

When compared to traditional multithreading, collaborative multithreading trades fairness for correctness. In traditional multithreading, preemption is the norm. It is easy to achieve fairness, because the system takes care of it through time slices. However, it is difficult to achieve correctness, because race conditions can arise virtually in any point of a program. With collaborative multithreading, or coroutines, there are no race conditions and therefore it is much easier to ensure correctness. However, the programmer must deal with fairness explicitly, by ensuring that long threads yield regularly.

Lua also offers multithreading by removing shared memory. In this case, the programming model follows Unix processes, where independent lines of execution do not share any kind of state: Each Lua process has its own logical memory space

[2] Simula offered coroutines for this reason [27].

with independent garbage collection. All communication is done through some form of message passing. Messages cannot contain references, because references (addresses) have no meaning across different processes. A main advantage of multiple processes is the ability to benefit from multi-core machines and true concurrency. Processes do not interfere with each other unless they explicitly request communication.

Lua does not offer an explicit mechanism for multiple processes, but it allows us to implement one as a library on top of stock Lua. Again, the book *Programming in Lua* [4] presents a simple implementation of a library for processes in Lua written with 200 lines of C code.

The key feature in Lua to allow such implementation is the concept of a *state*. Lua is an embedded language, designed to be used inside other applications. Therefore, it keeps all its state in dynamically-allocated structures, so that it does not interfere with other data from the application. If a program creates multiple Lua states, each one will be completely independent of the others.

The implementation of Lua processes uses multiple C threads, each with its own private Lua state. The library itself, in the C level, must handle threads, locks, and conditions. But Lua code that uses the library does not see that complexity. What it sees are independent Lua states running concurrently, each with its own private memory. The library provides also some communication mechanism. When two processes exchange data, the library copies the data from one Lua state to the other.

Currently there are two public libraries with such support: LuaLanes [28], which uses tuple spaces for communication, and Luaproc [29], which uses named channels.

7 Final Remarks

Lua is a small and simple language, but is also quite flexible. In particular, we have seen how it supports different paradigms, such as functional programming, object-oriented programming, goal-oriented programming, and data description.

Lua supports those paradigms not with many specific mechanisms for each paradigm, but with few general mechanisms, such as tables (associative arrays), first-class functions, delegation, and coroutines. Because the mechanisms are not specific to special paradigms, other paradigms are possible too. For instance, AspectLua [30] uses Lua for aspect-oriented programming.

All Lua mechanisms work on top of a standard procedural semantics. This procedural basis ensures an easy integration among those mechanisms and between them and the external world; it also makes Lua a somewhat conventional language. Accordingly, most Lua programs are essentially procedural, but many incorporate useful techniques from different paradigms. In the end, each paradigm adds important items into a programmer toolbox.

References

1. Millington, I.: Artificial Intelligence for Games. Morgan Kaufmann, San Francisco (2006)
2. Associação Brasileira de Normas Técnicas: Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital, ABNT NBR 15606-2 (2007)
3. Hempel, R.: Porting Lua to a microcontroller. In: de Figueiredo, L.H., Celes, W., Ierusalimschy, R. (eds.) Lua Programming Gems. Lua.org (2008)
4. Ierusalimschy, R.: Programming in Lua. second edn. Lua.org, Rio de Janeiro, Brazil (2006)
5. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: Lua 5.1 Reference Manual. Lua.org, Rio de Janeiro, Brazil (2006)
6. de Moura, A.L., Ierusalimschy, R.: Revisiting coroutines. ACM Transactions on Programming Languages and Systems 31(2), 6.1–6.31 (2009)
7. Ousterhout, J.K.: Scripting: Higher level programming for the 21st century. IEEE Computer 31(3) (March 1998)
8. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: The evolution of Lua. In: Third ACM SIGPLAN Conference on History of Programming Languages, San Diego, CA, June 2007, pp. 2.1–2.26 (2007)
9. Lamport, L.: LaTeX: A Document Preparation System. Addison-Wesley, Reading (1986)
10. Koyama, T., et al.: Simulation tools for damping in high frequency resonators. In: 4th IEEE Conference on Sensors, pp. 349–352. IEEE, Los Alamitos (2005)
11. Ueno, Y., Arita, M., Kumagai, T., Asai, K.: Processing sequence annotation data using the Lua programming language. Genome Informatics 14, 154–163 (2003)
12. Kelsey, R., Clinger, W., Rees, J.: Revised^5 report on the algorithmic language Scheme. Higher-Order and Symbolic Computation 11(1), 7–105 (1998)
13. Nehab, D.: Filters, sources, sinks and pumps, or functional programming for the rest of us. In: de Figueiredo, L.H., Celes, W., Ierusalimschy, R. (eds.) Lua Programming Gems, pp. 97–107. Lua.org (2008)
14. Steele Jr., G.L.: Rabbit: A compiler for Scheme. Technical Report AITR-474, MIT, Cambridge, MA (1978)
15. Adams, N., Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J.: ORBIT: an optimizing compiler for Scheme. SIGPLAN Notices 21(7) (July 1986); (SIGPLAN 1986)
16. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: The implementation of Lua 5.0. Journal of Universal Computer Science 11(7), 1159–1176 (2005); (SBLP 2005)
17. Ungar, D., Smith, R.B.: Self: The power of simplicity. SIGPLAN Notices 22(12), 227–242 (1987); (OOPLSA 1987)
18. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. SIGPLAN Notices 21(11), 214–223 (1986); (OOPLSA 1986)
19. Griswold, R., Griswold, M.: The Icon Programming Language. Prentice-Hall, New Jersey (1983)
20. Clocksin, W., Mellish, C.: Programming in Prolog. Springer, Heidelberg (1981)
21. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming 2(3), 323–343 (1992)
22. de Moura, A.L., Rodriguez, N., Ierusalimschy, R.: Coroutines in Lua. Journal of Universal Computer Science 10(7), 910–925 (2004); (SBLP 2004)
23. Haynes, C.T.: Logic continuations. J. Logic Programming 4, 157–176 (1987)

24. Wand, M., Vaillancourt, D.: Relating models of backtracking. In: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, Snowbird, UT, pp. 54–65. ACM, New York (2004)
25. Ierusalimschy, R., de Moura, A.L.: Some proofs about coroutines. Monografias em Ciência da Computação 04/08, PUC-Rio, Rio de Janeiro, Brazil (2008)
26. Ousterhout, J.K.: Why threads are a bad idea (for most purposes). In: USENIX Technical Conference (January 1996)
27. Birtwistle, G., Dahl, O., Myhrhaug, B., Nygaard, K.: Simula Begin. Petrocelli Charter (1975)
28. Kauppi, A.: Lua Lanes — multithreading in Lua (2009), `http://kotisivu.dnainternet.net/askok/bin/lanes/`
29. Skyrme, A., Rodriguez, N., Ierusalimschy, R.: Exploring Lua for concurrent programming. In: XII Brazilian Symposium on Programming Languages, Fortaleza, CE, August 2008, pp. 117–128 (2008)
30. Fernandes, F., Batista, T.: Dynamic aspect-oriented programming: An interpreted approach. In: Proceedings of the 2004 Dynamic Aspects Workshop (DAW 2004), March 2004, pp. 44–50 (2004)

Constraint Based Strategies

Claude Kirchner[1], Florent Kirchner[2], and Hélène Kirchner[1]

[1] INRIA
Centre de Recherche
INRIA Bordeaux - Sud-Ouest
351, cours de la Libération,
33405 Talence Cedex France
[2] INRIA
Centre de Recherche
INRIA Rennes - Bretagne Atlantique
Campus universitaire de Beaulieu,
35042 Rennes Cedex France

Abstract. Numerous computational and deductive frameworks use the notion of strategy to guide reduction and search space exploration, making the macro scale control of micro operations an explicit object of interest. In recent works, abstract strategies have been defined in extension but also intensionally. In this paper we complete these views with a new declarative approach based on constraints, which are used to model the different parts of a strategy. This procedure allows us to express elaborate strategies in a declarative and reusable way.

1 Introduction

The fundamental complementarity between deduction and computation, as emphasized in particular in deduction modulo [8], gives now rise to a completely new generation of proof assistants where customized deductions are performed modulo appropriate and user definable computations. This collusion of deduction and computation in next-generation proof assistants has inspired our recent work at providing a uniform definition for strategies, starting from a rule-based view point [12].

For term rewriting, reduction strategies study which expressions should be selected for evaluation and which rules should be applied. These choices are usually made to increase the efficiency of evaluation but may affect fundamental properties of computations such as confluence or (non-)termination. Programming languages like TOM [3], ELAN [4], Maude [20] and Stratego [22] allow for the explicit definition of the evaluation strategy.

Deductive environments include interactive proof assistants, automated theorem provers, proof checkers, and logical frameworks. For these systems, strategies (also called tacticals in some contexts) are used for proof search and proof planning, restriction of search spaces, specification of control components, combination of different proof techniques and computation paradigms, or meta-level programming in reasoning systems. Interest for these fundamental elements of deductive systems has been growing in recent years: see for instance [7,10,6,17] in the field of procedural proof assistants.

One main difficulty in defining strategies, both for deduction or computation, is precisely the specification of the strategy that the user wants the system to apply. Indeed, a strategy describes paths in the search or the reduction space that are complex objects, depending of the initial goal, but also of past and future of the current state exploration. To deal with this difficulty, we have first defined abstract strategies *in extension* [12] so that the strategy object, as well as its basic properties, can be studied thoroughly and precisely. A second step then consisted in providing a language to help describe the strategies of interest in an operational fashion. This approach is developed in [5] under the name of *intensional strategies*.

In what follows we develop another point of view, using the modeling capabilities of constraints to provide a declarative way of defining strategies. One of the main interest of this new approach is that the constraints solving process is taken care of by the environment, releasing the user of making procedural description of search or reduction paths. The constraints then become very powerful tools, well-suited to model the complexity, non-determinism, and infiniteness of strategies. By calling upon constraints either symbolic or numeric, the user can focus on the properties and correctness of the strategies he wants to define. Moreover, this declarative approach being free from circumstantial details such as an execution context, it bears the promise of both portability and reusability.

Based on the previous works of [12,5], the next section synthesizes the frameworks of abstract reduction strategies and of intensional strategies. We then define derivation schemas, a way to algebraize the notion of abstract strategies. Derivation variables are in particular introduced and are one of the main schematisation tool. These schemas can then be instantiated by the solutions of appropriate constraints via the adapted notion of substitution. We illustrate our approach by using progressively complex examples, making use mainly in this paper of equality constraints.

We assume the reader familiar with term rewriting and basic notions of term algebra as typically defined in [2] or [13].

2 Abstract Reduction Strategies

In this section, we recall from previous works, the general notion of abstract reduction systems already presented in [21,12,5], as well as the definitions of abstract strategies given in [12] and intensional strategies given in [5].

2.1 Abstract Reduction Systems

When abstracting the notion of strategies, one important preliminary remark is that we need to start from an appropriate notion of *abstract reduction system*

(ARS) based on the notion of oriented labeled graph instead of binary relation. This is due to the fact that, speaking of derivations, we need to make a difference between "being in relation" and "being connected". Typically modeling ARS as relations as in [2] allows us to say that, *e.g.*, a and b are in relation but not that there may be several different ways to derive b from a. Consequently, we need to use a more expressive approach, similar to the one proposed in [21, Chapter 8], based on a notion of oriented graph. Our definition is similar to the one given in [12] with the slight difference that we make more precise the definition of steps and labels. Similarly to the step-based definition of an abstract reduction system of [21], this definition that identifies the reduction steps avoids the so-called *syntactic accidents* [18], related to different but indistinguishable derivations.

Definition 1 (Abstract reduction system). *Given a countable set of objects $\mathcal{O}$ and a countable set of labels $\mathcal{L}$ mutually disjoint, an* abstract reduction system *(ARS) is a triple $(\mathcal{O}, \mathcal{L}, \Gamma)$ such that Γ is a functional relation from $\mathcal{O} \times \mathcal{L}$ to $\mathcal{O}$: formally, $\Gamma \subseteq \mathcal{O} \times \mathcal{L} \times \mathcal{O}$ and $(a, \phi, b_1) \in \Gamma$ and $(a, \phi, b_2) \in \Gamma$ implies $b_1 = b_2$.*

The tuples $(a, \phi, b) \in \Gamma$ are called steps *and are often denoted by $a \xrightarrow{\phi} b$. We say that a is the* source *of $a \xrightarrow{\phi} b$, b its* target *and ϕ its* label*. Moreover, two steps are* composable *if the target of the former is the source of the latter.*

The condition that Γ is a functional relation implies that an ARS is a particular case of a labeled transition system. Actually, labels characterize the way an object is transformed: given an object and a transformation, there is at most one object resulting from the transformation applied to this particular object.

The next definitions can be seen as a renaming of usual ones in graph theory. Their interest is to allow us to define uniformly derivations and strategies in different contexts.

Definition 2 (Derivation). *Given an abstract reduction system $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$ we call* derivation *over $\mathcal{A}$ any sequence π of steps $\big((t_i, \phi_i, t_{i+1})\big)_{i \in \Im}$ for any right-open interval $\Im \subseteq \mathbb{N}$ starting from 0. If $\Im$ contains at least one element, then:*

- *$Dom(\pi) = t_0$ is called the* source *(or domain) of π,*
- *$l(\pi) = (\phi_i)_{i \in \Im}$ is a sequence called* label *of π,*
- *For any non empty subinterval $\Im' \subseteq \Im$, $\pi' = \big((t_i, \phi_i, t_{i+1})\big)_{i \in \Im'}$ is a factor of π. If $\Im'$ contains 0, then π' is a prefix of π. If $\Im' \neq \Im$, π' is a strict factor (prefix).*

If $\Im$ is finite, it has a smallest upper bound denoted by $n_{\Im}$ or simply n and then:

- *$Im(\pi) = t_n$ is called the* target *(or image) of π,*
- *$|\,\pi\,| = card(\Im)$ is called the* length *of π,*

In such a case, π is said to be finite and is also denoted by $\pi = (t_0, l(\pi), t_n)$ or $t_0 \xrightarrow{l(\pi)} t_n$. The sequence containing no step is called empty derivation *and is denoted by Λ and by convention $l(\Lambda) = \epsilon$ where ϵ is the empty sequence of elements of $\mathcal{L}$.*

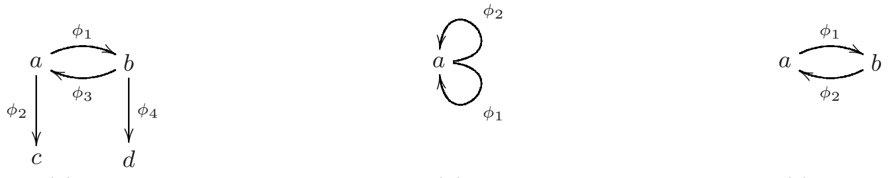
We denote by $\Gamma^{\omega}_{\mathcal{A}}$ (resp. $\Gamma^{*}_{\mathcal{A}}$, resp. $\Gamma^{+}_{\mathcal{A}}$) the set of all derivations (resp. finite, resp. non-empty and finite) over $\mathcal{A}$.

Note that a step may be considered as a derivation of length 1.

Definition 3 (Composable derivations). *Two derivations over a same abstract reduction system $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$, say $\pi_1 = \big((t_i, \phi_i, t_{i+1})\big)_{i \in \Im_1}$ and $\pi_2 = \big((u_i, \phi_i, u_{i+1})\big)_{i \in \Im_2}$ are* composable *iff either one of the derivation is empty or $\Im_1$ is finite and then $t_{n_1} = u_0$ where n_1 is the smallest upper bound of $\Im_1$. In such a case, the composition of π_1 and π_2 is the unique derivation $\pi = \big((v_i, \phi_i, v_{i+1})\big)_{i \in \Im}$ denoted by $\pi = \pi_1\pi_2$ such that for all $j < |\,\pi_1\,|$, $v_j = t_j$ and for all $j \geq |\,\pi_1\,|$, $v_j = u_{j-|\,\pi_1\,|}$.*

The composition is associative and has a neutral element which is Λ. Adopting the product notation, we denote $\prod_{i=1}^{n} \pi_i = \pi_1 \ldots \pi_n$, $\pi^n = \prod_{i=1}^{n} \pi$ and $\pi^{\omega} = \prod_{i \in \mathbb{N}} \pi$.

Like for graphs and labeled transition systems, in order to support intuition, we will often use the obvious graphical representation to denote the corresponding ARS.



(a) $\mathcal{A}_{lc}$ (b) $\mathcal{A}_{c}$ (c) $\mathcal{A}_{loop}$

Fig. 1. Graphical representation of abstract reduction systems

Example 1 (Abstract reduction systems). The abstract reduction system

$$\mathcal{A}_{lc} = (\{a, b, c, d\}, \{\phi_1, \phi_2, \phi_3, \phi_4\}, \{(a, \phi_1, b), (a, \phi_2, c), (b, \phi_3, a), (b, \phi_4, d)\})$$

with a finite number of objects but infinite derivations is depicted in Figure 1(a). $\Gamma^{\omega}_{\mathcal{A}_{lc}}$ contains for instance $\pi_1, \pi_1\pi_3, \pi_1\pi_4, \pi_1\pi_3\pi_1, (\pi_1\pi_3)^n, (\pi_1\pi_3)^{\omega}, \ldots$, with $\pi_1 = (a, \phi_1, b)$, $\pi_2 = (a, \phi_2, c)$, $\pi_3 = (b, \phi_3, a)$, $\pi_4 = (b, \phi_4, d)$.

Another abstract reduction system with an infinite set of objects and a countable infinite set of derivations starting from a same source is $\mathcal{A}_{ex} =$

$$(\{a_i^j \mid i, j \in \mathbb{N}\}, \mathbb{N} \cup \{\phi_i^j \mid 1 \leq i < j\}, \{(a_0^0, j, a_1^j) \mid 1 \leq j\} \cup \{(a_i^j, \phi_i^j, a_{i+1}^j) \mid 1 \leq i < j\})$$

whose relation can be depicted by:

$$\begin{tikzcd}
 & a_1^1 & & & & \\
 & a_1^2 \arrow[r, "\phi_1^2"] & a_2^2 & & & \\
a_0^0 \arrow[uur, "1"] \arrow[ur, "2"] \arrow[dr, "n"] \arrow[ddr, "\vdots"] & \vdots & & & & \\
 & a_1^n \arrow[r, "\phi_1^n"] & \cdots \arrow[r, "\phi_{n-2}^n"] & a_{n-1}^n \arrow[r, "\phi_{n-1}^n"] & a_n^n & \\
 & \vdots & & & &
\end{tikzcd}$$

2.2 Abstract Strategies

We use a general definition slightly different from the one used in [21, Chapter 9]. This approach has already been proposed in [12] and improved in [5].

Definition 4 (Abstract Strategy). *Given an ARS $\mathcal{A}$, an* abstract strategy *ζ over $\mathcal{A}$ is a subset of derivations of $\Gamma_{\mathcal{A}}^{\omega}$.*

A strategy can be a finite or an infinite set of derivations, and the derivations themselves can be finite or infinite in length.

An abstract strategy over an abstract reduction system $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$ induces a (partial) function from $\mathcal{O}$ to $2^{\mathcal{O}}$. This functional point of view has been already proposed in [4]; we just briefly recall it in our formalism.

The *domain* of a strategy ζ is the set of objects that are source of a derivation in ζ:

$$Dom(\zeta) = \bigcup_{\pi \in \zeta} Dom(\pi)$$

The application of a strategy is defined (only) on the objects of its domain. The application of a strategy ζ on $a \in Dom(\zeta)$ is denoted ζa and is defined as the set of all objects that can be reached from a using a finite derivation in ζ:

$$\zeta a = \{Im(\pi) \mid \pi \in \zeta, \pi \text{ finite and } Dom(\pi) = a\}$$

If $a \notin Dom(\zeta)$ we say that ζ *fails* on a (either ζ contains no derivation or it contains no derivation of source a).

If $a \in Dom(\zeta)$ and $\zeta a = \varnothing$, we say that the strategy ζ is *indeterminate* on a. In fact, ζ is indeterminate on a if and only if ζ contains no finite derivation starting from a.

Example 2 (Strategies). Let us consider again the abstract reduction system $\mathcal{A}_{lc}$ presented in Example 1 and define the following strategies:

1. The strategy $\zeta_u = \Gamma^{\omega}_{\mathcal{A}_{lc}}$, also called the *Universal* strategy [12] (*w.r.t.* $\mathcal{A}_{lc}$), contains all the derivations of $\mathcal{A}_{lc}$. We have $\zeta_u a = \zeta_u b = \{a, b, c, d\}$ and ζ_u fails on c and d.

2. The strategy $\zeta_f = \varnothing$, also called *Fail*, contains no derivation and thus fails on any $x \in \{a, b, c, d\}$.
3. For the strategy $\zeta_c = \left\{ \left(a \xrightarrow{\phi_1 \phi_3} a\right)^n a \xrightarrow{\phi_2} c \mid n \geq 0 \right\}$ no matter which derivation is considered, the object a eventually reduces to c: $\zeta_c a = \{c\}$. ζ_c fails on b, c and d.
4. The strategy $\zeta_\omega = \left\{ \left(a \xrightarrow{\phi_1 \phi_3} a\right)^\omega \right\}$ is not terminating on a and fails on b, c and d.

The so-called *Universal* and *Fail* strategies introduced in Example 2 can be obviously defined over any abstract reduction system.

2.3 Intensional Strategies

In [5], the notion of intensional strategy is introduced. The essence of the idea is that strategies are considered as a way of constraining and guiding the steps of a reduction. So at any step in a derivation, it should be possible to say whether a contemplated next step obeys the strategy ζ. In order to take into account the past derivation steps to decide the next possible ones, the history of a derivation has to be memorized and available at each step. Let us first introduce the notion of traced-object where each object memorizes how it has been reached.

Definition 5 (Traced-object). *Given a countable set of objects $\mathcal{O}$ and a countable set of labels $\mathcal{L}$ mutually disjoint, a* traced-object *is a pair $[\alpha]\, a$ where α is a sequence of elements of $\mathcal{O} \times \mathcal{L}$ called* trace *or* history*.*

The set of traces may be considered as a monoid $\left((\mathcal{O} \times \mathcal{L})^*, \odot\right)$ generated by $(\mathcal{O} \times \mathcal{L})$ and whose neutral element is denoted by Λ.

Definition 6 (Traced object compatible with an ARS). *A traced-object $[\alpha]\, a$ is* compatible *with $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$ iff $\alpha = ((a_i, \phi_i))_{i \in \Im}$ for any right-open interval $\Im \subseteq \mathbb{N}$ starting from 0 and $a = a_n$ and for all $i \in \Im$, $(a_i, \phi_i, a_{i+1}) \in \Gamma$. In such a case, we denote by $[\![\alpha]\!]$ the derivation $\left((a_i, \phi_i, a_{i+1})\right)_{i \in \Im}$ and by $\mathcal{O}^{[\mathcal{A}]}$ the set of traced objects compatible with $\mathcal{A}$. Moreover, we define an equivalence relation $\sim$ over $\mathcal{O}^{[\mathcal{A}]}$ as follows: $[\alpha]\, a \sim [\alpha']\, a'$ iff $a = a'$. We naturally have $\mathcal{O}^{[\mathcal{A}]} / \sim = \mathcal{O}$.*

An intensional strategy, as defined in [5], chooses the next step not only regarding the current object (or state), but also taking the history of objects into account.

Definition 7 (Intensional strategy (with memory)). *An* intensional strategy *over an abstract reduction system $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$ is a partial function λ from $\mathcal{O}^{[\mathcal{A}]}$ to 2^{Γ} such that for every traced object $[\alpha]\, a$, $\lambda([\alpha]\, a) \subseteq \{\pi \in \Gamma \mid Dom(\pi) = a\}$.*

An intensional strategy naturally generates an abstract strategy, as follows [5].

Definition 8 (Extension of an intensional strategy). *Let λ be an intensional strategy over an abstract reduction system $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$. The* extension *of λ is the abstract strategy ζ_λ consisting of the following set of derivations:*

$$\pi = ((a_i, \phi_i, a_{i+1}))_{i \in \Im} \in \zeta_\lambda \qquad \textit{iff} \qquad \forall j \in \Im, \quad (a_j, \phi_j, a_{j+1}) \in \lambda([\alpha]\, a_j)$$

where $\alpha = ((a_i, \phi_i))_{i \in \Im}$.

This extension may obviously contain infinite derivations; in such a case it also contains all the finite derivations that are prefixes of the infinite ones. Indeed, it is easy to see from Definition 8 that the extension of an intensional strategy is closed under taking prefixes.

It has been shown in [5] that the set of finite derivations generated by an intensional strategy λ can be constructed inductively as follows. Given an intensional strategy with memory λ over an abstract reduction system $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$, the finite support of its extension is the subset of ζ_λ made of only finite derivations. This is again an abstract strategy denoted $\zeta_\lambda^{<\omega}$ inductively defined as follows:

- $\forall\, [\Lambda]\, a \in \mathcal{O}^{[\mathcal{A}]}, \lambda([\Lambda]\, a) \subseteq \zeta_\lambda^{<\omega}$,
- $\forall \alpha$ *s.t.* $\pi = [\![\alpha]\!] \in \zeta_\lambda^{<\omega}$ and $\pi' \in \lambda\left([\alpha]\, Im(\pi)\right),\ \pi\pi' \in \zeta_\lambda^{<\omega}$

Each intensional strategy λ over $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$ induces an ARS $\mathcal{A}_\lambda = \left(\mathcal{O}^{[\mathcal{A}]}, \mathcal{L}, \Gamma_\lambda\right)$ such that $([\alpha]\, a, \phi, [\alpha \odot (a, \phi)]\, b) \in \Gamma_\lambda$ *iff* $(a, \phi, b) \in \lambda([\alpha]\, a)$. This is denoted by: $[\alpha]\, a \xrightarrow{\phi}_\lambda [\alpha \odot (a, \phi)]\, b$.

A special case are memoryless strategies, where the function λ does not depend on the history of the objects. This is the case of many strategies used in rewriting systems, as shown in the next examples.

Example 3. Let us define the following strategies:

- The intensional strategy λ_u defined on all objects in $\mathcal{O}$ such that for any object $a \in \mathcal{O}$, $\lambda_u(a) = \{\pi \mid \pi \in \Gamma, Dom(\pi) = a\}$ obviously generates the Universal strategy ζ_u (of Example 2).
- The intensional strategy λ_f defined on no object in $\mathcal{O}$ generates the Fail strategy ζ_f (of Example 2).
- Let us consider an abstract reduction system $\mathcal{A}$ where objects are terms, reduction of Γ is term rewriting with a rewrite rule in the rewrite system, and labels are positions where the rewrite rules are applied. Let us consider an order $<$ on the labels which is the prefix order on positions. Then the intensional strategy that corresponds to innermost rewriting is λ_{inn} such that $\lambda_{inn}(t) = \{\pi : t \xrightarrow{p} t' \mid p = max(\{p' \mid t \xrightarrow{p'} t' \in \Gamma\})\}$. When a lexicographic order is used, the classical *rightmost-innermost* strategy is obtained.

However the following examples of strategies cannot be expressed without the knowledge of the history and illustrate the interest of traced objects.

Example 4.

- The intensional strategy that restricts the derivations to be of bounded length k makes use of the size of the trace α, denoted $|\alpha|$:

$$\lambda_{ltk}([\alpha]\, a) = \{\pi \mid \pi \in \Gamma, Dom(\pi) = a, |\alpha| < k - 1\}$$

- The strategy that alternates reductions from a set (of steps) Γ_1 with reductions from a set Γ_2 can be generated by the following intensional strategy:

$$\begin{aligned}\lambda_{\Gamma_1;\Gamma_2}([\Lambda]\, a) &= \{\pi_1 \mid \pi_1 \in \Gamma_1, Dom(\pi_1) = a\} \\ \lambda_{\Gamma_1;\Gamma_2}([\alpha' \odot (u, \phi')]\, a) &= \{\pi_1 \mid \pi_1 \in \Gamma_1, Dom(\pi_1) = a\} \text{ if } u \xrightarrow{\phi'} a \in \Gamma_2 \\ \lambda_{\Gamma_1;\Gamma_2}([\alpha' \odot (u, \phi')]\, a) &= \{\pi_2 \mid \pi_2 \in \Gamma_2, Dom(\pi_2) = a\} \text{ if } u \xrightarrow{\phi'} a \in \Gamma_1\end{aligned}$$

However, as noticed in [5], the fact that intensional strategies generate only prefix closed abstract strategies prevents us from computing abstract strategies that look straightforward like the ones in the next example.

Example 5. We consider again the abstract reduction system $\mathcal{A}_{lc}$ and a strategy reduced to only one derivation $\zeta = \{a \xrightarrow{\phi_1} b \xrightarrow{\phi_3} a \xrightarrow{\phi_2} c\}$. ζ cannot be computed by an intensional strategy λ built as before since $\zeta_\lambda^{<\omega}$ would contain too many derivations, namely all prefixes of the derivation:

$$[\Lambda]\, a \xrightarrow{\phi_1}_\lambda [(a, \phi_1)]\, b \xrightarrow{\phi_3}_\lambda [(a, \phi_1) \odot (b, \phi_3)]\, a \xrightarrow{\phi_2}_\lambda [(a, \phi_1) \odot (b, \phi_3) \odot (a, \phi_2)]\, c$$

In a similar way, there is no intensional strategy that can generate a set of derivations of length exactly k.

The next section introduces another approach to define abstract strategies that in particular avoids this problem.

3 Constraints and Strategies

Since a strategy is a set of derivations, this set can be described in extension, as done in Example 2. But such definitions are not so convenient especially when the set is infinite. In this section, we make use of constraints as the basic language construction to describe strategies in a compact way. The schematization power of constraints is universally used in informatics and mathematics. More specifically, constraints have been extensively used in constraint logic programming [9], rewriting and deduction [14], and constraint solving itself [11] to mention just a few.

Since abstract reduction systems may involve infinite sets of objects, of reduction steps and of derivations, we can use constraints at different levels that can be ultimately combined: (i) to describe the objects occurring in a derivation (ii) to describe via the labels the requirements on the steps of reductions (iii) to describe the structure of the derivation itself (iv) to express requirements on the histories.

The framework we develop now allows us to define a strategy ζ as all instances $\sigma(S)$ of a derivation schema S such that σ is solution of a constraint C involving derivation variables in $\mathcal{X}_{\mathcal{D}}$, typically denoted by D, object variables in $\mathcal{X}_{\mathcal{O}}$, typically denoted X, and label variables in $\mathcal{X}_{\mathcal{L}}$, typically denoted L. Similar schemes could be done for history but are left out of this paper for simplification purpose. Therefore, in order to represent the objects is a generic way, we make use of (open) first-order terms $\mathcal{T}(\mathcal{F}, \mathcal{X}_{\mathcal{O}})$ over a signature $\mathcal{F}$.

Definition 9. *A* derivation schema *is a derivation on an abstract reduction system $\mathcal{A} = (\mathcal{T}(\mathcal{F}, \mathcal{X}_{\mathcal{O}}), \mathcal{L} \cup \mathcal{X}_{\mathcal{L}}, \Gamma \cup \mathcal{X}_{\mathcal{D}})$. It is a sequence of steps, written either $\pi = ((a_i, \phi_i, a_{i+1}))_{i \in \Im}$ or $\pi = a_j \xrightarrow{\phi_j} a_{j+1} \xrightarrow{\phi_{j+1}} \dots a_i \xrightarrow{\phi_i} a_{i+1}$ when π is finite, and where each a_i is a term of $\mathcal{T}(\mathcal{F}, \mathcal{X}_{\mathcal{O}})$, each ϕ_i may be a label or a variable in $\mathcal{X}_{\mathcal{L}}$, each step (a, ϕ, b) may be a step of Γ or a variable in $\mathcal{X}_{\mathcal{D}}$, each sequence π may be a sequence of steps or a variable in $\mathcal{X}_{\mathcal{D}}$.*

Notice that in this definition, we may also define term based labels and even derivations. Again, we choose not to get too general in this paper to keep the main ideas more apparent.

Notice also that, up to the labels, derivation schema are terms build over the binary symbol $\rightarrow$ with terms of $\mathcal{T}(\mathcal{F}, \mathcal{X}_{\mathcal{O}})$ as leaves. As we will see, this view is quite fruitful and require to assume from now on that the labeled arrow symbols are "associative" with neutral element Λ, in the following sense:

$$a_1 \xrightarrow{\phi_1} (a_2 \xrightarrow{\phi_2} a_3) = (a_1 \xrightarrow{\phi_1} a_2) \xrightarrow{\phi_2} a_3 \tag{1}$$

$$\Lambda \xrightarrow{\phi} a = a \xrightarrow{\phi} \Lambda = a \tag{2}$$

We have now to make clear how derivation schema can be instantiated to express derivations.

Definition 10. *Given a* derivation schema *S on an abstract reduction system $\mathcal{A} = (\mathcal{T}(\mathcal{F}, \mathcal{X}_{\mathcal{O}}), \mathcal{L} \cup \mathcal{X}_{\mathcal{L}}, \Gamma \cup \mathcal{X}_{\mathcal{D}})$, a substitution σ is composed of substitutions $\sigma_{\mathcal{O}}$ of $\mathcal{T}(\mathcal{F}, \mathcal{X}_{\mathcal{O}})$, $\sigma_{\mathcal{L}}$ from $\mathcal{X}_{\mathcal{L}}$ to $\mathcal{L} \cup \mathcal{X}_{\mathcal{L}}$ and σ_{Γ} from $\mathcal{X}_{\mathcal{D}}$ to $\Gamma \cup \mathcal{X}_{\mathcal{D}}$. The image of S by σ is given by the instance of each of its step:*

if $a, b \in \mathcal{T}(\mathcal{F}, \mathcal{X}_{\mathcal{O}})$, $\phi \in \mathcal{L} \cup \mathcal{X}_{\mathcal{L}}$, $D, D' \in \Gamma \cup \mathcal{X}_{\mathcal{D}}$, $\pi = ((a_i, \phi_i, a_{i+1}))_{i \in \Im}$,

$$\begin{aligned}
\sigma(a, \phi, b) &= (\sigma_{\mathcal{O}}(a) \xrightarrow{\sigma_{\mathcal{L}}(\phi)} \sigma_{\mathcal{O}}(b)) \\
\sigma(D, \phi, b) &= (\sigma_{\Gamma}(D) \xrightarrow{\sigma_{\mathcal{L}}(\phi)} \sigma_{\mathcal{O}}(b)) \\
\sigma(a, \phi, D) &= (\sigma_{\mathcal{O}}(a) \xrightarrow{\sigma_{\mathcal{L}}(\phi)} \sigma_{\Gamma}(D)) \\
\sigma(D, \phi, D') &= (\sigma_{\Gamma}(D) \xrightarrow{\sigma_{\mathcal{L}}(\phi)} \sigma_{\Gamma}(D')) \\
\sigma(\pi) &= (\sigma(a_i, \phi_i, a_{i+1}))_{i \in \Im}
\end{aligned}$$

To describe sets of derivations, it is convenient to use notations like $\{x \to D | D \in \zeta\}$ where by convention $x \to D$ is exactly x when $\zeta = \varnothing$.

Definition 11. *A* derivation constraint *over* $(\mathcal{T}(\mathcal{F}, \mathcal{X}_{\mathcal{O}}), \mathcal{L} \cup \mathcal{X}_{\mathcal{L}}, \Gamma \cup \mathcal{X}_{\Gamma})$ *is a formula* C *involving free variables variables of* $\mathcal{X}_{\mathcal{O}}$, $\mathcal{X}_{\mathcal{L}}$ *and* $\mathcal{X}_{\Gamma}$. *We denote* $Sol(C)$ *the set of solutions* σ *of* C*, that are ground instances of variables of* C*.*

The previous definition is on purpose quite general and leaves open the language on which the formulae are built. This can be for instance equality constraints, membership constraints or ordering constraints but also anti-pattern constraints like in [15] as well as first-order constraints or higher-order constraints involving functional variables, all these possibly occurring modulo some congruence typically defined by an equational theory like associativity or commutativity.

In order to make some syntactic evidence that a given formula is considered as a constraint, the predicate symbols appearing in such a constraint are distinguished with a question mark like in $X + Y =^? Z + X$.

Definition 12. *The abstract strategy schematized by* $(S \mid C)$*, where* S *is a derivation schema and* C *a derivation constraint over an abstract reduction system* $\mathcal{A} = (\mathcal{O} \cup \mathcal{X}_{\mathcal{O}}, \mathcal{L} \cup \mathcal{X}_{\mathcal{L}}, \Gamma \cup \mathcal{X}_{\Gamma})$*, is the subset* ζ *of derivations of* $\Gamma^{\omega}_{(\mathcal{O},\mathcal{L},\Gamma)}$ *defined as*

$$\zeta = \{\sigma(S) \mid \sigma \in Sol(C)\}$$

Note that for any (ground) abstract reduction system $\mathcal{A} = (\mathcal{O}, \mathcal{L}, \Gamma)$, the universal strategy, which corresponds to the set of all derivations can be (trivially) described as $(D \mid D \in^? \Gamma^{\omega}_{\mathcal{A}})$.

In order to relate these definitions with the previous concepts, we may point out that if we consider the abstract strategy ζ_λ generated by the intensional strategy λ, ζ_λ can be described as

$$(D \mid D \in^? \Gamma^{\omega}_{\mathcal{A}} \wedge [(X, L, Y) \in^? D \Rightarrow \ (X, L, Y) \in^? \lambda(X)])$$

To illustrate the expressivity of strategies with constraints, let us begin with an example of an abstract reduction system on terms, and remind the classical fact that if the symbol $\cdot$ is associative, then the equation $x \cdot a =^?_A a \cdot x$ has an infinite set of solutions $\{x \mapsto a^n | n \in N\}$ (where as usual we use the notation $a^n = a \cdot a^{n-1}$, a is a constant and x is a variable).

Example 6 (Simple constraints on terms). The infinite set of derivations of length one that transform a into $f(a^n)$ is simply described by:

$$(a \to f(X) \mid X \cdot a =^?_A a \cdot X)$$

But it is of course also interesting to use constraints to describe the structure of derivations. In this context a simple and useful remark is that, as mentioned above, the arrow symbols in derivation can be considered as associative, so that the derivations $a \to (b \to c)$ and $(a \to b) \to c$ are equivalent.

Example 7 (Simple constraints on derivations). Let us consider the ARS $\mathcal{A}_{lc}$ of Example 1. Then, the same equation as above allows us to describe ζ_{loop_1}, the infinite set of derivations that have a finite cycle on $a \to b$:

$$(D \mid D \xrightarrow{\phi_3} (a \xrightarrow{\phi_1} b) =^?_A (a \xrightarrow{\phi_1} b) \xrightarrow{\phi_3} D)$$

that is (omitting obvious labels for simplicity)

$$\{a \to b, (a \to b) \to (a \to b), (a \to b)^3, \ldots\} = \{(a \to b)^n \forall n \geq 0\}.$$

Since we have no restriction on linearity, we may do it twice as in

$$(D \xrightarrow{L} D \mid D \xrightarrow{L} (a \xrightarrow{\phi_1} b) =^?_A (a \xrightarrow{\phi_1} b) \xrightarrow{\phi_3} D)$$

(notice here the use of the label variable).

Then strategies like $\zeta_c = \left\{ \left(a \xrightarrow{\phi_1 \phi_3} a \right)^n \xrightarrow{\phi_2} c \mid n \geq 0 \right\}$ given in Example 2, can be simply expressed is the following way:

$$(D \xrightarrow{\phi_3} a \xrightarrow{\phi_2} c \mid D \xrightarrow{\phi_3} (a \xrightarrow{\phi_1} b) =^?_A (a \xrightarrow{\phi_1} b) \xrightarrow{\phi_3} D).$$

Example 8. In order to insert an element a at all possible positions into a given derivation π, we can use again an equational constraint modulo associativity and write :

$$(D \to a \to D' \mid D \to D' =^?_A \pi).$$

Example 9. To illustrate more complex constraints, let us express the set of derivations $\mathcal{A}_{ex}$ of Example 1 as follows:

$$(a^0_0 \xrightarrow{L} X \xrightarrow{L'} D \mid X \in^? \mathcal{O} \wedge L \in^? \mathbb{N} \wedge L' \in^? \mathcal{L} \wedge |X \to D| =^? L \wedge D \in^? \Gamma^*_{\mathcal{A}_{ex}})$$

where $|X \to D|$ denotes the length of this derivation.

Constraints can also be used to control exponents in regular expressions or be used to impose precise behaviors, e.g., for all variables to provide different values. We may use this agility by introducing anti-patterns [16].

Let us terminate this list of increasingly elaborated examples with a constraint based description of innermost rewriting.

Example 10 (Innermost). In this example, we assume the reader familiar with first-order term rewriting. The arrow symbol here means rewriting with a set of rewrite rules $\mathcal{R}$ build over a set of term $T(F, X)$. Therefore, given a term, a derivation step is characterized by an occurrence and a rewrite rule. How can we describe, using appropriate constraints, that only innermost redexes are reduced? One of the difficulty is that the property to be innermost is not intrinsic to a derivation but to the application of a derivation to a term.

Let x be a variable representing the term to be reduced. The set of innermost derivations can be defined in the following way:

$$\begin{array}{lll}
IM(x) = \big(D \mid & (\exists y \in T(F,X), D =^? x \to D' & \text{the derivation is non} \\
& \wedge & \text{ empty} \\
& D' \in^? IM(y) & \text{the remainder should} \\
& \wedge & \text{ be innermost too} \\
& (\exists g \to d \in \mathcal{R}, \exists \omega, \exists \sigma, x_{|\omega} =^? \sigma(g)) & \text{it matches} \\
& \wedge & \\
& \neg(\exists w' < w, \exists l \to r \in \mathcal{R}, \exists \alpha, x_{|\omega'} =^? \alpha(l)) & \text{with no match above} \\
& \wedge & \\
& y =^? x[\sigma(d)]_\omega)\big) & \text{gives the value to } y.
\end{array}$$

Remark that when a term is normalized, there is no IM derivation. As for the variety of the language, notice the set constraint on the second part of the conjunction. Finally, we could also use matching constraints to get rid of the explicit use of σ and α.

Relating this definition to the intensional strategy λ_{inn} defined in Example 3, we could also write:

$$\begin{array}{lll}
IM(x) = \big(D \mid & (\exists y,\ D =^? x \xrightarrow{L} D' & \text{the derivation is non empty} \\
& \wedge & \\
& D' \in^? IM(y) & \text{the remainder should be innermost too} \\
& \wedge & \\
& (x \xrightarrow{L} y) \in^? \lambda_{inn}(x))\big) & \text{with } \lambda_{inn} \text{ defined as in Example 3.}
\end{array}$$

4 Conclusion

In this paper we came back on the definition of abstract strategies. After recalling the extensional and intensional ways to define them, we introduced derivation schemas and constraint based description of strategies. We applied this to both simple and more complex examples to illustrate the agility of the constraint based approach.

This procedure is quite powerful and allows us to describe, in a declarative fashion, strategies than can arguably be seen as elaborate. Our goal here is to free the user from the burden of engaging in the procedural definition of search spaces or reduction trees, and the complexity associated with the specification of history- and future-dependent derivations.

We are planning to apply this approach to theorem proving strategies, such as focusing, that currently rely mainly on syntactic apparatus [1,19] to guide proof search exploration. Application to constraint based specification of reduction strategies also appears quite challenging and remains to be explored.

Let us mention that the ability to explicitly use negations or complement problems could be of great help is defining strategies. For instance, the possibility of excluding conflicting assignments from proof exploration is a feature that is widely used in modern SAT solvers. In this context, we anticipate anti-patterns [16] to provide a useful constrained strategy tool.

Let us finally remark that strategies are currently defined using ML-like functional languages in theorem provers like Coq, HOL or Isabelle. This work is a first step to allow for a constraint based family of strategy languages in a constraint programming style.

References

1. Andreoli, J.-M.: Logic Programming with Focusing Proofs in Linear Logic. Journal of Logic and Computation 2(3), 297–347 (1992)
2. Baader, F., Nipkow, T.: Term Rewriting and all That. Cambridge University Press, Cambridge (1998)
3. Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
4. Borovanský, P., Kirchner, C., Kirchner, H., Ringeissen, C.: Rewriting with strategies in ELAN: a functional semantics. International Journal of Foundations of Computer Science 12(1), 69–98 (2001)
5. Bourdier, T., Cirstea, H., Dougherty, D.J., Hélène, K.: Extensional and Intensional Strategies. In: Proceedings of the 9th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2009), Brasilia, June 28. Electronic Proceedings in Theoretical Computer Science (2010)
6. Coen, C.S., Tassi, E., Zacchiroli, S.: Tinycals: Step by Step Tacticals. Electronic Notes in Theoretical Computer Science 174(2), 125–142 (2007)
7. Delahaye, D.: A Tactic Language for the System Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 85–95. Springer, Heidelberg (2000)
8. Dowek, G., Hardin, T., Kirchner, C.: Theorem Proving Modulo. Journal of Automated Reasoning 31(1), 33–72 (2003)
9. Jaffar, J., Lassez, J.-L.: Constraint Logic Programming. In: Proceedings of the 14th Annual ACM Symposium on Principles Of Programming Languages, Munich, Germany, pp. 111–119 (1987)
10. Jojgov, G.: Holes with Binding Power. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 162–181. Springer, Heidelberg (2003)
11. Jouannaud, J.-P., Kirchner, C., Kirchner, H.: Incremental Construction of Unification Algorithms in Equational Theories. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 361–373. Springer, Heidelberg (1983)
12. Kirchner, C., Kirchner, F., Kirchner, H.: Strategic Computations and Deductions. In: Benzmüller, C., Brown, C.E., Siekmann, J., Statman, R. (eds.) Reasoning in Simple Type Theory. Festschrift in Honour of Peter B. Andrews on His 70th Birthday. Studies in Logic and the Foundations of Mathematics, vol. 17, pp. 339–364. College Publications (2008)
13. Kirchner, C., Kirchner, H.: Rewriting, Solving, Proving. A preliminary version of a book (1999), http://www.loria.fr/ckirchne/=rsp/rsp.pdf
14. Kirchner, C., Kirchner, H., Rusinowitch, M.: Deduction with symbolic constraints. Revue d'Intelligence Artificielle 4(3), 9–52 (1990); Special issue on Automatic Deduction

15. Kirchner, C., Kopetz, R., Moreau, P.-E.: Anti-Pattern Matching. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 110–124. Springer, Heidelberg (2007)
16. Kirchner, C., Kopetz, R., Moreau, P.-E.: Anti-Pattern Matching Modulo. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 275–286. Springer, Heidelberg (2008)
17. Kirchner, F., Muñoz, C.: The Proof Monad. Submitted to the Journal of Logic and Algebraic Programming (2008)
18. Lévy, J.-J.: Réductions correctes et optimales dans le lambda-calcul. PhD thesis, Université de Paris VII (1978)
19. Liang, C., Miller, D.: A Unified Sequent Calculus for Focused Proofs. In: LICS: 24th Symp. on Logic in Computer Science, pp. 355–364 (2009)
20. Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a Strategy Language for Maude. In: Martí-Oliet, N. (ed.) Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27–April 4, 2004. Electronic Notes in Theoretical Computer Science, vol. 117, pp. 417–441. Elsevier, Amsterdam (2005)
21. Bezem, M., Klop, J.W., de Vrijer, R. (eds.): Terese. Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
22. Visser, E.: Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001)

Integrating ILOG CP Technology into $\mathcal{TOY}$[*]

Ignacio Castiñeiras[1] and Fernando Sáenz-Pérez[2]

[1] Dept. Sistemas Informáticos y Computación
[2] Dept. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid
`ncasti@fdi.ucm.es, fernan@sip.ucm.es`

Abstract. The constraint functional logic programming system $\mathcal{TOY}$ has been using the SICStus Prolog finite domain ($\mathcal{FD}$) constraint solver. In this work, we show how to integrate the ILOG CP $\mathcal{FD}$ constraint solving technology into this system, with the aim of improving its application domain and performance. We describe our implementation emphasizing the synchronization between Herbrand computations in the $\mathcal{TOY}$ side and $\mathcal{FD}$ constraint solving in the ILOG CP side. Finally, performance results are reported and discussed.

1 Introduction

$\mathcal{TOY}$[1] is a system implemented in SICStus Prolog 3.12.8 [11]. Its operational semantics is based on a lazy narrowing calculus and includes several constraint domains allowing its cooperation. This system allows Herbrand equality and disequality constraints (managed by the constraint domain $\mathcal{H}$), linear and non-linear arithmetic constraints over reals ($\mathcal{R}$), finite domain constraints over integers ($\mathcal{FD}$), and a communication domain $\mathcal{M}$ which makes possible the cooperation among $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$. Whereas $\mathcal{R}$ as $\mathcal{FD}$ rely on the constraint solvers provided by SICStus Prolog, solving in $\mathcal{H}$ and $\mathcal{M}$ needs an explicit management [3]. $\mathcal{TOY}$ offers a wide range of finite domain constraints comparable to many CLP($\mathcal{FD}$) systems, using a concrete constraint solving system as one of its components [5]. Here, we focus on this particular constraint domain for integrating a new constraint solving system based on ILOG CP technology.

The generic component architecture of the connection between $\mathcal{TOY}$ and its external $\mathcal{FD}$ constraint system is shown to the left of Fig. 1. $\mathcal{TOY}$ identifies each $\mathcal{FD}$ constraint during goal solving, and factorizes this (possibly) composed contraint into primitive ones, adding new produced variables if necessary [3]. Then, it posts these primitive constraints to $solve^{FD}$, which acts as an intermediary between $\mathcal{TOY}$ and the external $\mathcal{FD}$ system. $solve^{FD}$ sends the constraints to this system and collects its computed answers.

[*] This work has been partially supported by the Spanish projects TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 and UCM-BSCH-GR58/08-910502.

1.1 $\mathcal{TOY}$ with SICStus Prolog CLP($\mathcal{FD}$): $\mathcal{TOY}(\mathcal{FD}s)$

$\mathcal{TOY}$ (referred to as $\mathcal{TOY}(\mathcal{FD}s)$ from now on) has been using the $\mathcal{FD}$ constraint system provided in the library `clpfd` of SICStus Prolog, which is basically composed of a constraint store and solver. The component architecture of the connection between $\mathcal{TOY}$ and SICStus Prolog $\mathcal{FD}$ constraint system is shown in the middle of Fig. 1. Next, we show a basic example for illustrating the use of the system $\mathcal{TOY}(\mathcal{FD}s)$ with finite domains constraints.

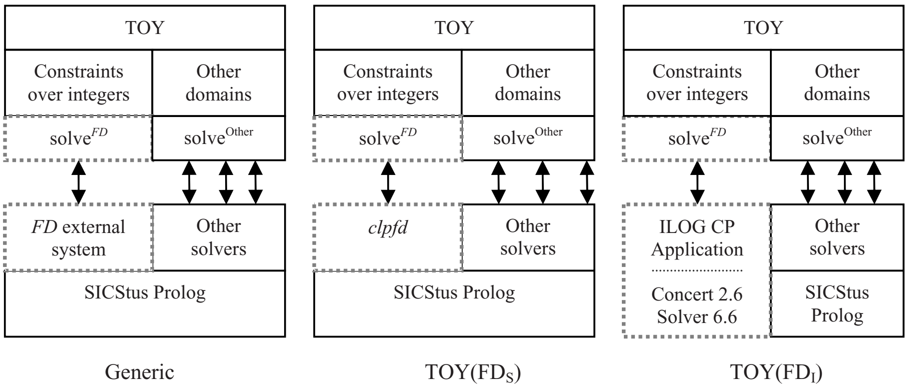


Fig. 1. Architectural Components

Example 1. Let's consider that `X` is an integer between `5` and `12`, `Y` is an integer between `2` and `17`, `X+Y=17` and `X-Y=5`. It is possible to solve this problem in $\mathcal{TOY}(\mathcal{FD}s)$ as shown in the following interactive session:

```
TOY(FDs)> X #>= 5, X #<= 12, Y #>= 2, Y #<= 17,
          X #+ Y == 17, X #- Y == 5
    yes
    { 5 # + Y #= X,
      X # + Y #= 17,
      X in 10..12,
      Y in 5..7 }
    Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
    no
    Elapsed time: 0 ms.
```

However, the use of the SICStus Prolog $\mathcal{FD}$ system reveals some disadvantages:

– Recent works [2] have proved that its performance can be enhanced, which is needed when dealing with complex problems.

- The constraint solver works as a black-box for predefined search processing. This precludes user-defined interactions for pruning the search tree.
- There are no debugging capabilities allowing, for instance, to derive the subset of infeasible constraints.

1.2 ILOG CP to Improve $\mathcal{TOY}$

ILOG CP 1.4 [7] is an industrial technology market leader. It has a declarative nature. It provides a C++ API to access its libraries. Its $\mathcal{FD}$ constraint solver works as a glass-box, allowing interactions during the solving process. It also includes debugging techniques helping the user to discover the unfeasible subset of the input constraint set. It allows the user to define new classes of constraints in order to formulate different and complex properties. The use of different constraint solvers for a unique application domain is also allowed. Moreover, libraries using specific, efficient algorithms for solving complex scheduling problems are provided.

Any ILOG CP 1.4 application isolates objects responsible of modeling the user problem from objects responsible of solving any concrete model. Following this idea, the problem is modeled in a generic language, easing the task of expressing the constraints of the problem. Once the modeling phase is completed, the model can be solved by one or more different constraint solvers. The solver extracts all of the modeling objects contained into the model, creating a one-to-one object translation. This new objects belonging to the solver are semantically equivalent to the modeling objects, but their internal structure is targeted at the solver. It is possible to access each object created by the solver through the associated object contained into the model. The most paradigmatic tool representing this philosophy is ILOG OPL Studio [8]. ILOG CP 1.4 includes the library ILOG Concert 2.6 to provide the necessary interface for connecting models to solvers. Three libraries are provided for $\mathcal{FD}$ constraint solving:

* ILOG Solver 6.6, for generic $\mathcal{FD}$ problems solving.
* ILOG Scheduler 6.6, with specific algorithms for solving scheduling problems.
* ILOG Dispatcher 4.6, with specific algorithms for solving routing problems.

In this work we will consider only ILOG Solver 6.6. Fig. 2. shows the basic objects needed to model and solve the $\mathcal{FD}$ problem proposed in Example 1:

- `IloEnv` ***env***. It manages the memory of any object of the application.
- `IloModel` ***model***`(env)`. Is the main modeling object. Contains the set of objects responsible of formulating the $\mathcal{FD}$ problem, which are:
  - $\mathcal{FD}$ constraints, each of them modeled as an `IloConstraint` object.
  - $\mathcal{FD}$ decision variables, each of them modeled as an `IloIntVar` object.
- `IloIntVarArray` ***vars***`(env)`. This vector is intended to make possible to reference, from a unique object, any `IloIntVar` contained in `model`.
- `IloSolver` ***solver***`(env)`. Is the main solving object. Contains the set of objects responsible of solving the $\mathcal{FD}$ problem, which are:
  - $\mathcal{FD}$ constraints, each of them modeled as an `IlcConstraint` object.
  - $\mathcal{FD}$ decision variables, each of them modeled as an `IlcIntVar` object.

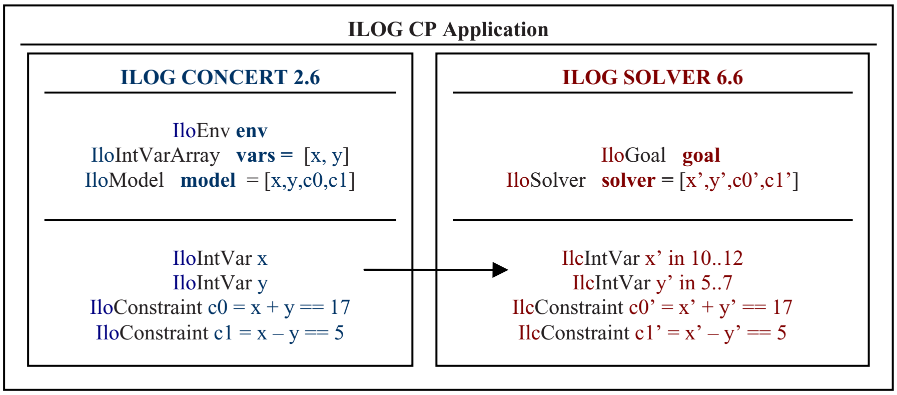

ILOG CP Application
ILOG CONCERT 2.6
IloEnv env
IloIntVarArray vars = [x, y]
IloModel model = [x,y,c0,c1]
IloIntVar x
IloIntVar y
IloConstraint c0 = x + y == 17
IloConstraint c1 = x – y == 5
ILOG SOLVER 6.6
IloGoal goal
IloSolver solver = [x',y',c0',c1']
IlcIntVar x' in 10..12
IlcIntVar y' in 5..7
IlcConstraint c0' = x' + y' == 17
IlcConstraint c1' = x' – y' == 5

Fig. 2. Generic ILOG CP Application

The main methods of `solver` we use in this work are:

- `solver.extract(model)`. For each `IloIntVar` and `IloConstraint` contained in `model` it creates an associated new `IlcIntVar` or `IlcConstraint` object, whose internal structure is targeted at `solver` solving techniques.
- `solver.propagate()`. Propagates the `IlcConstraint` set contained in `solver`. This propagation prunes some values of the `IlcIntVar` set contained in `solver` by using limit consistency techniques. In Fig. 2. we can see the remaining values of the `IlcIntVar` set contained in `solver` after the propagation of the `IlcConstraint` set.
- `solver.solve(goal)`. Uses the labeling enumeration procedure defined in `goal` to look for a first concrete solution of the $\mathcal{FD}$ problem contained in `solver`.

– `IloGoal` ***goal***`(env,vars,Strategy)`. This object represents a labeling enumeration procedure which labels the `IlcIntVars` contained in `solver` associated to the `IloIntVars` contained in `vars`. By this labeling procedure, `solver` is able to find the different extensional solutions to the $\mathcal{FD}$ problem. In this work we use two labeling strategies predefined in ILOG Solver 6.6:
  - A static search procedure `IloChooseFirstUnboundInt`, which selects the variables in the textual order they occur in `vars`.
  - A dynamic search procedure 'first fail' `IloChooseMinSizeInt`, which selects first the variable of `vars` with minimum domain size.

  For a given variable, both strategies select first the minimum value in its domain.

2 $\mathcal{TOY}$ with ILOG CP: $\mathcal{TOY}(\mathcal{FD}i)$

In this section, we explain in detail how to integrate ILOG CP $\mathcal{FD}$ technology into the system $\mathcal{TOY}$ (referred to as $\mathcal{TOY}(\mathcal{FD}i)$ from now on). $\mathcal{TOY}$ is implemented in SICStus Prolog while ILOG CP is a technology implemented and available in C++. So, first we study how to make a connection between $\mathcal{TOY}$ and ILOG CP by connecting SICStus Prolog and C++. Our approach is based on the integration of a C++ foreign resource into a SICStus Prolog application. Due to the different nature of both languages, we study the emerging difficulties to establish a communication between $\mathcal{TOY}$ and ILOG CP, as well as the decisions we have made to solve them. Also, an example of the behavior of the new system $\mathcal{TOY}(\mathcal{FD}i)$ is shown.

2.1 Connecting SICStus Prolog to C++

It is possible to communicate a SICStus Prolog application to a C++ component. The C++ component needs to be a dynamic library with a specific internal file structure. This communication is done by mapping a set of linking Prolog predicates (contained in the Prolog application) to a set of C++ functions (defined in the C++ component). SICStus Prolog also defines a set of possible conversions between Prolog arguments and C++ arguments. Each argument of a linking Prolog predicate must also indicate if it is either an input argument (sent to the C++ function) or an output argument (computed by the C++ function). There is a bidirectional conversion between a Prolog term and the C++ type `SP_term_ref`. By invoking `SP_term_ref` object methods, C++ functions can perform the following actions:

– Create and assign Prolog terms.
– Obtain the contents of a Prolog term.
– Compare and unify Prolog terms.

This context supports the necessary conditions to connect $\mathcal{TOY}$ and ILOG CP by making just a few changes in the component architecture of $\mathcal{TOY}$, whose new structure can be seen on the right hand side of Fig. 1.

- From the point of view of $\mathcal{TOY}$, it is necessary to put a new Prolog predicate in any place of $solve^{FD}$ where a communication with ILOG CP is needed (posting a new constraint, declaring a new ILOG decision variable, etc.)
- On the other hand, we build a new ILOG CP application which integrates ILOG Concert 2.6 and ILOG Solver 6.6 libraries. This application contains instances of the basic modeling and solving objects explained in Section 1.2. It also includes the set of C++ functions linked to the existing Prolog predicates in $solve^{FD}$.

Each time $solve^{FD}$ calls any interfaced predicate, first, it turns all Prolog arguments into C++ arguments. Next, it transfers the program control to the C++ function, which uses and/or computes them within its body. Once the C++ function has finished, the execution control comes back to $solve^{FD}$, which continues with the evaluation of the next call.

Creating a SICStus Prolog C++ Foreign Resource

SICStus Prolog supplies a tool, `splfr` [10], for creating a dynamic library as, for instance `interface.dll` taking as input two files:

- `interface.pl` Declares the mapping of each Prolog predicate to each C++ function. It groups all of these functions in a unique resource. For example:
  ```
  foreign(f1,p1(+integer)).
  foreign(f2,p2(+term,-term)).
  foreign_resource(interface,[f1,f2]).
  ```
- `interface.cpp` Includes the C++ functions mapped to Prolog facts. It adds as many auxiliary functions and libraries as needed. For example:
  ```
  void f1(long l){...}
  void f2(SP_term_ref t1, SP_term_ref t2){...}
  ```

The macro `splfr` is used as a shortcut to the execution of some compiling and linking commands offered by Microsoft Visual C++ [9]. First of all, taking `interface.pl` as input, it creates two new files, `interface_glue.c` and `interface_glue.h`, which provides the necessary glue code for the SICStus application.

2.2 Communication between $\mathcal{TOY}$ and ILOG CP

In this section we explain in detail how to solve the communication difficulties between SICStus Prolog and ILOG CP in the system $\mathcal{TOY}(\mathcal{FD}i)$. As $\mathcal{TOY}$ is a system implemented in SICStus Prolog, the communication between $\mathcal{TOY}$ and its $\mathcal{FD}$ technology is quite natural in $\mathcal{TOY}(\mathcal{FD}s)$. However, as ILOG CP is implemented in C++, some glue code is needed to fix the impedance mismatch problem in $\mathcal{TOY}(\mathcal{FD}i)$.

There have been four difficult tasks to overcome in the new system $\mathcal{TOY}(\mathcal{FD}i)$. We explain each of them in the next subsections. When we make reference to any ILOG CP application object, we use the notation of Section 1.2. To this end, we use `model` if we refer to the ILOG Concert 2.6 model object, we use `solver` if we refer to the ILOG Solver 6.6 generic $\mathcal{FD}$ solver, and we use `vars` if we refer to the decision variables contained in `model`.

Managing $\mathcal{FD}$ constraints

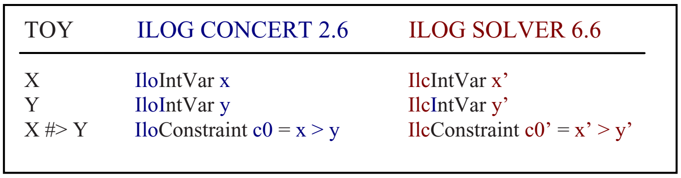
The set of $\mathcal{FD}$ constraints of a $\mathcal{TOY}$ goal involves a set of logic variables that we denote as '$\mathcal{FD}$ logic variables'. To model the $\mathcal{FD}$ constraint set with ILOG CP, some points must be taken into account:

- We need to create as many `IloIntVar` decision variables as $\mathcal{FD}$ logic variables take part into the $\mathcal{FD}$ constraint set. These variables must be added to `model` and `vars`.
- We must find a bijective relation that associates each $\mathcal{FD}$ logic variable of the $\mathcal{TOY}$ goal to each decision variable existing in the ILOG CP vector `vars`.
- We model each $\mathcal{FD}$ constraint in ILOG CP over the set of decision variables of the vector `vars` associated to the set of $\mathcal{FD}$ logic variables involved in that $\mathcal{FD}$ constraint.

Whatever way of communication between $\mathcal{TOY}$ and ILOG CP, for each $\mathcal{FD}$ constraint and each $\mathcal{FD}$ logic variable we need three instances:

- The $\mathcal{FD}$ constraint and $\mathcal{FD}$ logic variable contained in $\mathcal{TOY}$.
- The `IloConstraint` and `IloIntVar` contained in `model`.
- The `IlcConstraint` and `IlcIntVar` created by `solver` from its associated `IloConstraint` and `IloIntVar` contained in `model`, respectively.

Fig. 3. shows the association between the different instances of an element in $\mathcal{TOY}$, ILOG Concert 2.6 and ILOG Solver 6.6.



TOY	ILOG CONCERT 2.6	ILOG SOLVER 6.6
X	IloIntVar x	IlcIntVar x'
Y	IloIntVar y	IlcIntVar y'
X #> Y	IloConstraint c0 = x > y	IlcConstraint c0' = x' > y'

Fig. 3. Association between $\mathcal{TOY}$ and ILOG CP

A first attempt for mapping a $\mathcal{FD}$ logic variable to a decision variable of `vars` is tried. It intends to manage `vars` and a `SP_term_ref` vector, making them evolve simultaneously. The elements of the `SP_term_ref` vector are in fact the `SP_term_ref` conversions of the $\mathcal{FD}$ logic variables. Each time $solve^{FD}$ sends a new $\mathcal{FD}$ constraint to ILOG CP, the associated C++ function will first look for its $\mathcal{FD}$ logic variables in the `SP_term_ref` vector. If it can not find any variable, we can ensure that the C++ function is dealing with a new $\mathcal{FD}$ logic variable not handled before. So, the C++ function adds this new $\mathcal{FD}$ logic variable to the `SP_term_ref` vector last position, say `i`. Immediately, a new `IloIntVar` decision variable is created and added to `model` and `vars[i]`. When each $\mathcal{FD}$ logic variable of the $\mathcal{FD}$ constraint sent by $solve^{FD}$ is contained at an index of the `SP_term_ref` vector, the $\mathcal{FD}$ constraint is modeled over the decision variables of `vars` associated to these indexes.

However, this first attempt fails. This is due to the rules which govern the scope of a `SP_term_ref`. When a C++ function containing `SP_term_refs` (as arguments or dynamically created within it) finishes its execution, all these `SP_term_refs` become invalid. Let's see the next example, where we define an interface between the Prolog predicates `p1`, `p2` and `p3` and the C++ functions `f1`, `f2` and `f3`, respectively. Functions `f1` and `f2` receive a Prolog term as an argument, while `f3` receives two Prolog terms.

– Let's call `p3` with two occurrences of the logic variable `X`, as `p3(X,X)`. If we call `SP_compare(t1,t2)` within `f3(SP_term_ref t1, SP_term_ref t2)` the result says that both `SP_term_refs` are in fact the same Prolog term.
– But, let's do the call `p1(X)`. We store `t1` of `f1(SP_term_ref t1)` in a global vector with type `SP_term_ref`. When `f1` finishes, the program control comes back to Prolog. Now, we call `p2` with the logic variable `X` again, `p2(X)`. If we call `SP_compare(t1,t2)` within `f2(SP_term_ref t2)` between `t2` and the `SP_term_ref` stored in the vector during `f1`, the result says that both

`SP_term_ref`s are different. There is no doubt that both are in fact the same Prolog term. The problem is that, when `f1` finishes, the `SP_term_ref` stored in the vector becomes invalid.

The second and successful attempt relies on the management of the bijective relation, which is done in the Prolog application by the use of a list of $\mathcal{FD}$ logic variables (referred to as `V` from now on). We want `V` to be used in each $solve^{FD}$ predicate. On the one hand, SICStus Prolog does not allow global variables. On the other hand, there is a logic variable `Cin` [4], which represents a mixed constraint store and is common to each $solve^{FD}$ predicate. Our plan is to store any data structure demanded by the communication between $\mathcal{TOY}$ and ILOG CP, specifically our $\mathcal{FD}$ logic variables list `V`, into `Cin`. Each time a $solve^{FD}$ predicate manages a new $\mathcal{FD}$ constraint, we can check whether a $\mathcal{FD}$ logic variable belongs to `V` or not by accessing it within `Cin`. Any new $\mathcal{FD}$ logic variable is automatically added to the end of `V`, say at position `i`. Here, a new call to the C++ function which creates a new `IloIntVar` is done. This function adds this decision variable to `model` and `vars[i]`. Once all $\mathcal{FD}$ logic variables of the $\mathcal{FD}$ constraint belong to `V`, $solve^{FD}$ determines their indexes, and puts them as arguments to the C++ function, which models the $\mathcal{FD}$ constraint by adding to `model` a new `IloConstraint` over the associated positions of `vars`.

Synchronizing ILOG CP with $\mathcal{TOY}$

$\mathcal{TOY}$ can also bind its $\mathcal{FD}$ logic variables through an equality constraint in the Herbrand solver. For example, in the goal `TOY(FDi)> X #>= 0, X == 3` the variable `X` is bound to the value `3`. This is done by the Prolog term unification which results from the Herbrand equality constraint `X == 3`. This unification is visible at any occurrence of that $\mathcal{FD}$ logic variable, particularly the one in `V`. This causes an inconsistency between the contents of `V` and `vars`. To repair this lack of synchronization we must send an equality constraint to ILOG CP, making the mapped decision variable in `vars` equals to the bound value.

A first attempt tries to synchronize by an event-driven approach. To capture events, SICStus Prolog provides the module of attributed variables. This module assigns attributes to a set of logic variables. Each time an attributed logic variable is bound, the predicate `verify_attributes(+Var, +Value, +Goals)` is triggered. We use the attribute `fd` for each $\mathcal{FD}$ logic variable. Thus, each time the Herbrand solver binds a $\mathcal{FD}$ logic variable, `verify_attributes(+Var, +Value, +Goals)` will automatically call the C++ function which synchronizes the associated decision variable of `vars`.

However, this first attempt fails. For this synchronization we need to know which index does the associated decision variable have in `vars`. We can only get this index by looking for the $\mathcal{FD}$ logic variable in `V`. But the arguments of `verify_attributes(+Var, +Value, +Goals)` are fixed. As SICStus Prolog does not allow global variables, there is no way to get access to `V`.

A second attempt consists of making the Herbrand solver responsible of calling the C++ synchronization function. But this idea must be rejected, because there is a basic principle of independency between the different solvers of the system $\mathcal{TOY}$. Any solution to this problem must respect the idea of solving the synchronization within $solve^{FD}$.

The third (and successful) attempt modifies the internal structure of `V`. Now it becomes a list of pairs. The first element of each pair contains the $\mathcal{FD}$ logic variable, and the second one contains a flag which determines if the bound $\mathcal{FD}$ logic variable has been synchronized with `vars`. Thus, while the $\mathcal{FD}$ logic variable is not bound, the value of the flag remains at `0`. When the $\mathcal{FD}$ logic variable becomes bound, the value of the flag indicates whether the variable of `vars` is synchronized or not.

Each time $solve^{FD}$ sends a new $\mathcal{FD}$ constraint to ILOG CP, it must previously:

– Look for any pair in `V` (say at position `i`) whose pattern is `[`*`value`*`,0]`
– Add to `model` the new `IloConstraint` `vars[i]==`*`value`*.
– Change the pair at position `i` of `V` by `[`*`value`*`,1]`

Once there are no pairs with the pattern `[`*`value`*`,0]` in the list, $solve^{FD}$ is able to send the new $\mathcal{FD}$ constraint. Also, at the end of the $\mathcal{TOY}$ goal, a new synchronization is done. This synchronization attempt is clearly inefficient, making it a task to be improved in new releases of $\mathcal{TOY}(\mathcal{FD}i)$. Let's consider the next goal:

```
Toy(FDi)> X #>= 2, X == 1, X1 == 1, X2 == 1, ... , X1000 == 1
```

The first $\mathcal{FD}$ logic variable of the goal in the narrowing order is `X`, which occurs at the first position of `V` and `vars`. The synchronization of `X == 1` as `vars[0] == 1` makes the $\mathcal{FD}$ problem infeasible. So, the $\mathcal{TOY}$ goal will fail after `X == 1`, and there is no need for computing the rest of the goal expressions. However, the first equality `vars[0] == 1` is not computed until the next $\mathcal{FD}$ constraint is posted.
As `X == 1, X1 == 1, X2 == 1, ... , X1000 == 1` are computed by the Herbrand solver there are no more $\mathcal{FD}$ constraints in the goal, so the synchronization will not occur until the end of the goal. The goal will useless compute a thousand of successful expressions. After that, it synchronizes `vars[0] == 1` and fails.

Synchronizing $\mathcal{TOY}$ with ILOG CP

ILOG CP can bind variables in `vars` via the `IlcConstraint` set propagation produced by `solver.propagate()` or a labeling enumeration procedure `goal` used by `solver.solve(goal)`. This produces a lack of synchronization between the vector `vars` and `V`. To synchronize, whenever any `IlcIntVar var'` (associated to the `IloIntVar var` contained in `vars[i]`) is bound to *value*, the pair `[`*`Var`*`,0]` contained at position `i` of `V` must be automatically unified with `[`*`value`*`,1]`.

To this end, the predicates of $solve^{FD}$ send the list `V` as an input argument to the C++ functions managing the $\mathcal{FD}$ constraints in ILOG CP. An output argument is also added to obtain the new state of `V` computed by the C++ function after `solver` propagation or labeling. A new global variable `vector<int,int>` must be created in ILOG CP. Each pair of the vector contains:

– The index `i` in `vars` of the `IloIntVar` associated to the `IlcIntVar` treated.
– The *value* that `solver` has obtained for this `IlcIntVar`.

Any C++ function clears `vector<int,int>` at the beginning of the managment of a new $\mathcal{FD}$ constraint. After the solving techniques used, the C++ function accesses to the content of `vector<int,int>`, to see whether there are any variable that has been bound. By using `vector<int,int>` and `V`, the C++ function builds the new state of `V`, unifying as many $\mathcal{FD}$ logic variables as `vector<int,int>` demands.

The only remaining task to be explained is how the solving techniques add automatically each pair to `vector<int,int>`. To do so, we use demons to capture bind events. Thus, a new demon object `IlcDemon RealizeVarBound` is created. It concerns on how to insert each new pair into the `vector<int,int>`. This demon is triggered by the propagation of a constraint `IlcCheckWhenBound`. Each `IlcCheckWhenBound` constraint involves one `IloIntVar`. This constraint propagates when the `IlcIntVar` associated to this `IloIntVar` becomes bound. ILOG CP associates a demon to a method of a constraint class. When the demon is triggered, the method of this constraint class is automatically executed. We associate `RealizeVarBound` to the method `varDemon` of the `IlcCheckWhenBound` constraint class. This method checks the index in `vars` of the associated `IloIntVar` of the bound `IlcIntVar` and its value, adding both of them as a new pair of integers to the global `vector<int,int>`. We summarize how our ILOG CP application adds the pairs to the `vector<int,int>` in the next three steps:

– For each new decision variable `IloIntVar` added to `vars[i]` and `model`, we impose the constraint `IlcCheckWhenBound`.
– When the `IlcIntVar` associated to `vars[i]` is bound to *value*, `IlcCheckWhenBound` propagates, triggering the demon `RealizeVarBound`.
– `RealizeVarBound` executes the `IlcCheckWhenBound` method `varDemon`, which adds the pair `<i,value>` to `vector<int,int>`.

Solutions in $\mathcal{TOY}(\mathcal{FD}i)$

Any $\mathcal{TOY}(\mathcal{FD}s)$ solution is expressed in general with constraints (equality, disequality and $\mathcal{FD}$ constraints –including ranges–). Of course, $\mathcal{TOY}(\mathcal{FD}s)$ accepts to label $\mathcal{FD}$ variables by using a $\mathcal{FD}$ labeling enumeration procedure, in order to obtain the extensional solution to a goal.

The system $\mathcal{TOY}(\mathcal{FD}i)$ presented in this work reproduces the solution structure of $\mathcal{TOY}(\mathcal{FD}s)$. But, as a first approach, we do not support the use of backtracking, so we can only use labeling enumeration procedures to look for the first concrete solution to a goal. When the goal is completely finished, we show to the user the set of non-ground $\mathcal{FD}$ constraints as well as the remaining values of the $\mathcal{FD}$ variables.

ILOG Solver 6.6 does not grant access to simplified constraints (i.e., solved forms). So, to show the solution to the user, we do not parse the `IlcConstraint` set contained in `solver`. Instead of that, we store in `Cin` a list with the $\mathcal{FD}$ constraints (referred to as `C` from now on) appearing in the $\mathcal{TOY}$ goal. When a *solve*^{FD} predicate manages a new $\mathcal{FD}$ constraint of the goal, this constraint is added to `C`. In the solution we show any non-ground element of `C`.

To show the remaining values of the $\mathcal{FD}$ logic variables, we access to each `IlcIntVar` contained in `solver` throughout its associated `IloIntVar` contained in `model`. ILOG Solver 6.6 provides some methods to check the remaining values of these variables.

2.3 A $\mathcal{TOY}(\mathcal{FD}i)$ Example

In this section we detail how goal solving works with the new system $\mathcal{TOY}(\mathcal{FD}i)$ following Example 1:

```
Toy(FDi)> X #>= 5, X #<= 12, Y #>= 2, Y #<= 17,
          X #+ Y == 17, X #- Y == 5
```

We specify how the data structures of *solve*^{FD} and ILOG CP evolve with each goal expression evaluation. On the one hand, we look at the state of `V` and `C` within `Cin`. On the other hand, we look at the state of `vars`, `model` and `solver` by pointing out any `IloIntVar`, `IloConstraint`, `IlcIntVar`, `IlcConstraint` object accessed through them. Any new element added by the evaluation of a goal expression is highlighted in boldface. At the beginning of the computation, all data structures are empty, as we can see in Fig. 4.

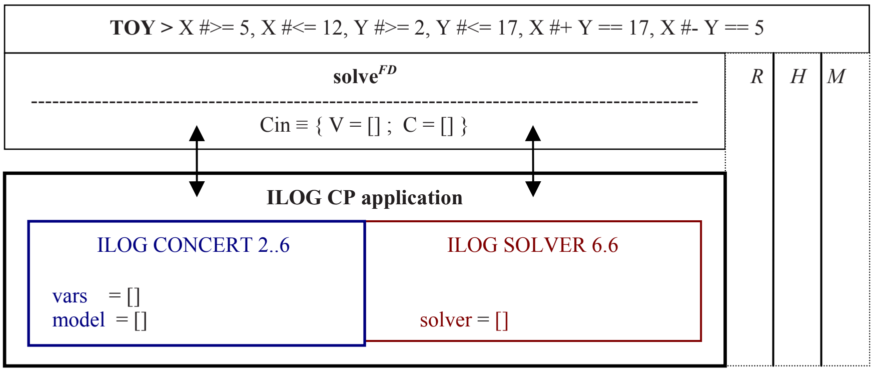


Fig. 4. Beginning of the computation

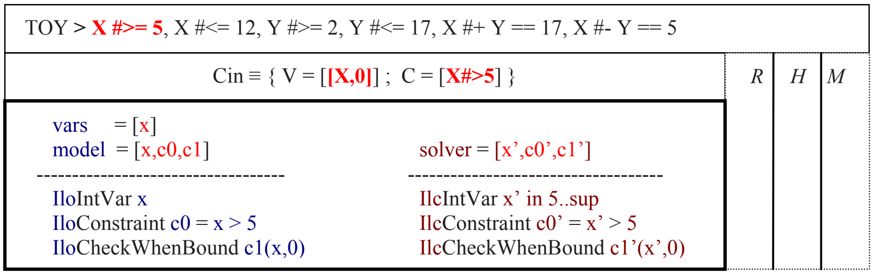
Now, we detail in Fig. 5. the evaluation of the goal expression `X #>= 5`. First, we model the $\mathcal{FD}$ constraint in ILOG Concert 2.6:

1. As `X` is not contained in `V`, we add the new pair `[X,0]` in the position 0 of `V`. Then, we create a new `IloIntVar x` and we add it to `vars[0]` and `model`. Now, `X` and `x` are associated due to they are at the same index (0) of `V` and `vars`, respectively.
2. We add the propagator `IloCheckWhenBound c1(x,0)` to be able to synchronize the $\mathcal{FD}$ logic variable `X` when the `IlcIntVar` associated to `x` becames bound to a value.

3. We add the new $\mathcal{FD}$ constraint `X #>= 5` to `C`. Then, we add to `model` the `IloConstraint c0`.

Then, we solve the $\mathcal{FD}$ problem contained in `model`:

4. We use `solver.extract(model)` to make a one-to-one object translation of the `model` content. This creates the new objects `x',c0'` and `c1'`.
5. We use `solver.propagation()`. It prunes some values of the domain of `x'`.
6. As the state of `solver` remains feasible, $\mathcal{TOY}$ continues with the evaluation of the next goal expression.



```
TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == 17, X #- Y == 5

Cin ≡ { V = [[X,0]] ;  C = [X#>5] }

vars   = [x]
model  = [x,c0,c1]                     solver = [x',c0',c1']
----------------------------------     -----------------------------------
IloIntVar x                            IlcIntVar x' in 5..sup
IloConstraint c0 = x > 5               IlcConstraint c0' = x' > 5
IloCheckWhenBound c1(x,0)              IlcCheckWhenBound c1'(x',0)
```

Fig. 5. Evaluation of the first $\mathcal{FD}$ constraint

We do not detail here the evaluation of `X #<= 12, Y #>= 2` and `Y# <= 17`, which are quite similar to `X #>=5`. The evaluation continues with the expression `X #+ Y == 17`. As this is a compound constraint, $\mathcal{TOY}$ decomposes it into the primitive constraints `X #+ Y == _Z` and `_Z == 17`.

- The evaluation of `X #+ Y == _Z` adds this constraint to `C`. It also adds `[_Z,0]` to `V`, `_z` to `model` and `_z'` to `solver`.
- In the evaluation of `_Z == 17` the constraint is sent to the Herbrand solver $\mathcal{H}$, which binds the variable `_Z` to the value 17. This causes that the instances of `_Z` in `V` and `C` are also unified to 17, producing a lack of consistency between `_Z` and its associated variables in ILOG `_z` and `_z'`. The synchronization of `_z` and `_z'` will happen with the management of the next $\mathcal{FD}$ constraint or at the end of the $\mathcal{TOY}$ goal.

The evaluation continues with the expression `X #- Y == 5`. As this is a compound constraint, $\mathcal{TOY}$ decomposes it into the primitive constraints:
`X #- Y == _T` and `_T == 5`. We detail here the evaluation of the goal expression `X #- Y == _T`, which can be seen in Fig. 6.

In the top of Fig. 6. we see the state before the evaluation starts. We can see that the constraint `_Z == 17` appears in the Herbrand solver of $\mathcal{TOY}$. Also, the instances of `_Z` in `V` and `C` are highlighted, because they have been unified to 17. The variables associated to `_Z` in ILOG are also highlighted, because they have not been bound yet to the value `17`.

In the middle of Fig. 6. we see the state after the synchronization of `_Z` with `_z` and `_z'`. Now the `IlcIntVar _z'` is also bound to the value `17`. The pair

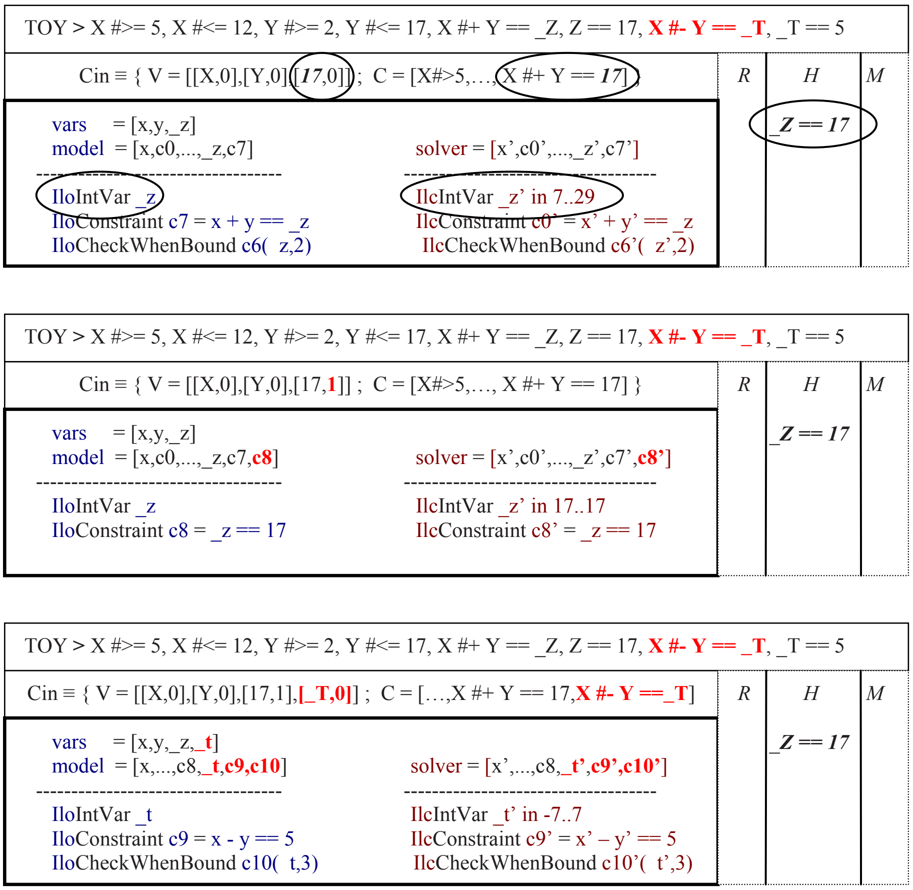

TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == _Z, Z == 17, X #- Y == _T, _T == 5
Cin ≡ { V = [[X,0],[Y,0],[17,0]] ; C = [X#>5,…, X #+ Y == 17] }
R
H
M
_Z == 17
vars = [x,y,_z]
model = [x,c0,...,_z,c7]
solver = [x',c0',...,_z',c7']
IloIntVar _z
IloConstraint c7 = x + y == _z
IloCheckWhenBound c6(z,2)
IlcIntVar _z' in 7..29
IlcConstraint c0' = x' + y' == _z
IlcCheckWhenBound c6'(z',2)
TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == _Z, Z == 17, X #- Y == _T, _T == 5
Cin ≡ { V = [[X,0],[Y,0],[17,1]] ; C = [X#>5,…, X #+ Y == 17] }
R
H
M
_Z == 17
vars = [x,y,_z]
model = [x,c0,...,_z,c7,c8]
solver = [x',c0',...,_z',c7',c8']
IloIntVar _z
IloConstraint c8 = _z == 17
IlcIntVar _z' in 17..17
IlcConstraint c8' = _z == 17
TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == _Z, Z == 17, X #- Y == _T, _T == 5
Cin ≡ { V = [[X,0],[Y,0],[17,1],[_T,0]] ; C = […,X #+ Y == 17,X #- Y ==_T]
R
H
M
_Z == 17
vars = [x,y,_z,_t]
model = [x,...,c8,_t,c9,c10]
solver = [x',...,c8,_t',c9',c10']
IloIntVar _t
IloConstraint c9 = x - y == 5
IloCheckWhenBound c10(t,3)
IlcIntVar _t' in -7..7
IlcConstraint c9' = x' – y' == 5
IlcCheckWhenBound c10'(t',3)

Fig. 6. Constraint management with synchronization

[17,0] of V has been changed by [17,1], because the variables associated in ILOG are now synchronized.

After synchronizing ILOG CP with the equalities produced by the Herbrand solver, we manage the constraint X #- Y == _T. We can see this in the bottom of Fig. 6.

$\mathcal{TOY}$ continues with the evaluation of the next goal expression. The constraint _T == 5 is sent to the Herbrand solver $\mathcal{H}$. This will bind the variable _T to the value 5. The instances of _T in V and C will be unified to 5. This produces a lack of consistency between _T (now bound to 5) and its associated variables in ILOG _t and _t'. As there are no more $\mathcal{FD}$ constraints, the synchronization will happen at the end of the $\mathcal{TOY}$ goal.

With this last synchronization we create a new IloConstraint c = _t == 5 in model. Then solver will translate and propagate this new constraint, binding the IlcIntVar _t'. We modify the pair [5,0] of the list V to [5,1].

Now the goal is completely finished. As the state of `solver` remains feasible, $\mathcal{TOY}$ shows the solution of the computation to the user. First we show the $\mathcal{FD}$ constraints by displaying any non-ground term contained in `C`. Then we show the values for the $\mathcal{FD}$ logic vars by accessing its associated `IlcIntVars` contained in ILOG. We do not show the extra variables produced during narrowing.

```
Toy(FDi)> X #>= 5, X #<= 12, Y #>= 2, Y #<= 17,
          X #+ Y == 17, X #- Y == 5
      yes
      { X #>= 5
        X #=< 12
        Y #>= 2
        Y #=< 17
        X #+ Y == 17
        X #- Y == 5
        X in 10..12
        Y in 5..7 }
      Elapsed time: 0 ms.
```

In Fig. 7. we see the state of $\mathcal{TOY}(\mathcal{FD}i)$ after the computation.

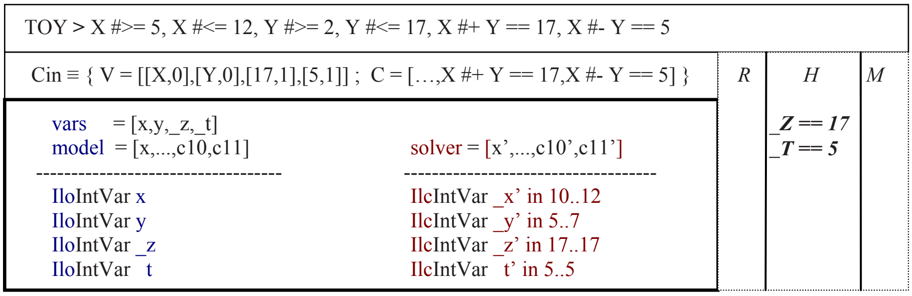


Fig. 7. $\mathcal{TOY}(\mathcal{FD}i)$ state after computation

3 Measuring Performance

In this section we use two test parametric, scalable (on `n`) benchmark programs which model systems of linear equations $A * X = b$. Each system has `n` independent equations with `n` variables `[X1,...,Xn]` whose domains are `{1..n}`. Each system has a unique integer solution. The matrix A takes the value i on its diagonal coefficients $A_{i,i}$ and the value 1 for the rest of them.

Both benchmark programs have been run in a machine with an Intel Dual Core 2.4Ghz processor and 4GB RAM memory. The SO used is Windows XP SP3. The SICStus Prolog version used is 3.12.8. The ILOG CP application used is ILOG CP 1.4, with ILOG Concert 2.6 and ILOG Solver 6.6 libraries. Microsoft Visual C++ 6.0. tools are used for compiling and linking the application.

We show performance results (expressed in miliseconds) for the following systems: both $\mathcal{TOY}(\mathcal{FD}s)$ and $\mathcal{TOY}(\mathcal{FD}i)$ just described, and also for a C++ program directly modelling the problems using the ILOG CP libraries (denoted by FDs, FDi and $ILOG$ in the tables, respectively). The latter will help us in analysing the overhead due to $\mathcal{TOY}$ implementation of lazy narrowing.

For each benchmark, we show three instances of `n`: 4, 12 and 15 variables. In each case, we present results for the two labeling strategies `IloChooseFirstUnboundInt` and `IloChooseMinSizeInt` (denoted by ff) presented in the section 1.2.

Also, we show the speedups of $\mathcal{TOY}(\mathcal{FD}i)$ with respect to $\mathcal{TOY}(\mathcal{FD}s)$ and ILOG CP respectively. Specifically, we denote as:

- (a) the speedup of $\mathcal{TOY}(\mathcal{FD}i)$ with respect to $\mathcal{TOY}(\mathcal{FD}s)$ using the static search procedure to solve the problem.
- (b) the speedup of $\mathcal{TOY}(\mathcal{FD}i)$ with respect to $\mathcal{TOY}(\mathcal{FD}s)$ using the 'first fail' search procedure.
- (c) the speedup of $\mathcal{TOY}(\mathcal{FD}i)$ with respect to ILOG CP C++ program using the static search procedure.
- (d) the speedup of $\mathcal{TOY}(\mathcal{FD}i)$ with respect to ILOG CP C++ program using the 'first fail' search procedure.

The benchmark programs are:

- First: The solution `[X1,...,Xn]` holds: $\forall i \in \{1 \ldots n\}$ `Xi` $= i$. Performance measurement gives the following results:

`n`	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
4	0	15	0	0	15	15	1.0	-	0	0
12	31	1.750	156	516	15	281	5.0	0.29	10.4	1.83
15	297	299,312	423	67,376	63	20,578	1.42	0.22	6.7	3.27

  For this first benchmark, $\mathcal{TOY}(\mathcal{FD}i)$ takes more time than $\mathcal{TOY}(\mathcal{FD}s)$ for solving with the static search procedure, but less time for the dynamic search procedure. The solving time difference between them grows as we increase the number of variables for the benchmarks. Looking at how the domains of the variables evolve after the initial constraint propagation, we can conclude that the structure of the solution for this first benchmark fits quite well into the static search procedure, while it is dramatically harmful to the dynamic search procedure. This help us to realize that, for problems where the needed exploration to obtain the solution is really small, then $\mathcal{TOY}(\mathcal{FD}i)$ is slower than $\mathcal{TOY}(\mathcal{FD}s)$. This is because of the time involved in the communication between the Prolog implementation of $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG CP. However, as the nodes needed to be explored increase slightly, this waste of time is overcome, making $\mathcal{TOY}(\mathcal{FD}i)$ more efficient than $\mathcal{TOY}(\mathcal{FD}s)$.
- Second: The solution `[X1,_,Xn]` holds: $\forall i \in \{1..n\}$ `Xi` $= n-(i-1)$. The above conclusions are clearly confirmed in this second benchmark, as $\mathcal{TOY}(\mathcal{FD}i)$ is faster than $\mathcal{TOY}(\mathcal{FD}s)$ for both search procedures. In this case, the structure of the solution is dramatically harmful for the static strategy, while it

behaves better for the dynamic strategy. In the former, $\mathcal{TOY}(\mathcal{FD}i)$ takes slightly less solving time than $\mathcal{TOY}(\mathcal{FD}s)$. In any case, these measurements point out that our first approach to integrate the ILOG CP technology into $\mathcal{TOY}(\mathcal{FD}i)$ is encouraging, but also that the management of the additional data structures used for the interface should be optimized. Performance measurement gives the following results:

n	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
4	16	16	16	31	31	15	1.0	1.93	0.51	2.06
12	531	250	437	126	109	63	0.83	0.50	4	2
15	15,563	21,968	13,937	3,406	843	1,765	0.90	0.16	16.53	1.93

4 Conclusions and Future Work

In this work, we have studied how to integrate the $\mathcal{FD}$ ILOG CP technology into the system $\mathcal{TOY}$. We have shown that this technology offers some advantages over the existing system $\mathcal{TOY}(\mathcal{FD}s)$ based on the $\mathcal{FD}$ technology of SICStus Prolog. We have described in detail our implementation, showing that the application architecture of $\mathcal{TOY}$ and ILOG CP are hard to integrate in terms of a correct communication between them. We have shown by means of two scalable benchmarks that the new system $\mathcal{TOY}(\mathcal{FD}i)$ is faster than $\mathcal{TOY}(\mathcal{FD}s)$ as the benchmark increases its size. However, we have concluded that there is a performance penalization due to the management of the data structures that make possible the connection of $\mathcal{TOY}$ with its new $\mathcal{FD}$ component. Therefore, optimizing this management will be the target of our immediate future work. As many practical $\mathcal{FD}$ problems covered by $\mathcal{TOY}(\mathcal{FD}s)$ require the use of non-deterministic functions, backtracking management will be covered in a next work. This will also allow us to use labeling enumeration procedures to find the different extensional solutions of a goal. So, we will be able to deal with an extended set of benchmarks (as the one seen in [5]) in $\mathcal{TOY}(\mathcal{FD}i)$ future releases. Another subject of interest is to test the constraint libraries ILOG Scheduler 6.6 and ILOG Dispatcher 4.6 bundled in ILOG CP 1.4, as well as other constraint libraries, as Gecode [6].

References

1. Arenas, P., Estévez, S., Fernández, A., Gil, A., López-Fraguas, F., Rodríguez-Artalejo, M., Sáenz-Pérez, F.: $\mathcal{TOY}$. A multiparadigm declarative language. version 2.3.1 (2007); Caballero, R., Sánchez, J. (eds.), http://toy.sourceforge.net
2. del Campo, R.G., Sáenz-Pérez, F.: Programmed search in a timetabling problem over finite domains. Electronic Notes in Theoretical Computer Science 177, 253–267 (2007)
3. Estévez-Martín, S., Fernández, A., Hortalá-González, M., Sáenz-Pérez, F., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: On the Cooperation of the Constraint Domains H, R and FD in $CFLP$. Theory and Practice in Logic Programming 9(4), 415–527 (2009)

4. Estévez-Martín, S., Fernández, A.J., Sáenz-Pérez, F.: About implementing a constraint functional logic programming system with solver cooperation. In: Proc. of CICLOPS 2007, pp. 57–71 (2007)
5. Fernández, A.J., Hortalá-González, T., Sáenz-Pérez, F., del Vado-Vírseda, R.: Constraint Functional Logic Programming over Finite Domains. Theory and Practice in Logic Programming 7(5), 537–582 (2007)
6. Gecode. Gecode, `http://www.gecode.org/`
7. ILOG. ILOG Solver 6.6, Reference Manual (2008)
8. ILOG. ILOG OPL Studio 6.1, Reference Manual (2009)
9. Microsoft (2005), `http://msdn.microsoft.com/en-us/visualc/default.aspx`
10. SICStus Prolog. Using SICStus Prolog with newer Microsoft C compilers, `http://www.sics.se/isl/sicstuswww/site/dontpanic.html`
11. SICStus Prolog (2007), `http://www.sics.se/isl/sicstus`

Termination of Context-Sensitive Rewriting with Built-In Numbers and Collection Data Structures[*]

Stephan Falke and Deepak Kapur

CS Department, University of New Mexico, Albuquerque, NM, USA
{spf,kapur}@cs.unm.edu

Abstract. Context-sensitive rewriting is a restriction of rewriting that can be used to elegantly model declarative specification and programming languages such as Maude. Furthermore, it can be used to model lazy evaluation in functional languages such as Haskell. Building upon previous work on constrained equational rewrite systems (CERSs), an expressive and elegant class of rewrite systems that contains built-in numbers and supports the use of collection data structures such as sets or multisets, context-sensitive rewriting with CERSs is investigated in this paper. This integration results in a natural way for specifying algorithms in the rewriting framework. In order to automatically prove termination of this kind of rewriting, a dependency pair framework for context-sensitive rewriting with CERSs is developed, resulting in a flexible termination method that can be automated effectively. Several powerful termination techniques are developed within this framework. An implementation in the termination prover AProVE has been successfully evaluated on a large collection of examples, including several examples obtained from functional Maude modules.

1 Introduction

Ordinary term rewrite systems (TRSs) have been succesfully used for modeling algorithms in a functional programming style. Ordinary TRSs, however, impose serious drawbacks. First, collection data structures such as sets or multisets cannot be represented easily since these non-free data structures typically cause non-termination of the ordinary rewrite relation. Notice that these collection data structures are used in real-life functional programming languages such as OCaml (using Moca [7], which adds relational data types to the language) and can be used in Maude by specifying suitable equational attributes. Second, and equally severe, domain-specific knowledge about primitive data types such as natural numbers or integers is not directly available in ordinary TRSs. These primitives are available in any real-life programming language, thus making an integration into the term rewriting framework highly desirable. It has been shown in [12] that constrained equational rewrite systems (CERSs) provide an expressive and

[*] Partially supported by NSF grants CCF-0541315 and CNS-0831462.

convenient tool for modeling algorithms that solves both of these drawbacks. Since [12] considers only natural numbers as a primitive data type, the first contribution of this paper is a reformulation of the ideas from [12] that allows for built-in integers.[1] An integration of integers into the term rewriting framework is important for automated termination proving since most currently available termination techniques are based on syntactic considerations, whereas termination of algorithms operating on integers often requires semantical reasoning that is commonly based on properties of $\geq$ or $>$ in integers.

Example 1. This example illustrates the use of collection data structures and integers in Maude and OCaml. Consider the following functional Maude module operating on lists.

```
fmod LISTS is
  protecting INT .
  sorts List .
  subsorts Int < List .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  op length : List -> Int .
  op reverse : List -> List .
  var N : Int .
  var K L : List .
  eq length(nil) = 0 .
  eq length(N) = 1 .
  eq length(K L) = length(K) + length(L) .
  eq reverse(nil) = nil .
  eq reverse(N) = N .
  eq reverse(K L) = reverse(L) reverse(K) .
endfm
```

The same program can also be written in OCaml (using Moca [7]).

```
type 'a t = private
  | Nil
  | Element of 'a
  | Concat of 'a t * 'a t
    begin
      associative
      neutral (Nil)
    end
let rec length x = match x with
  | Nil -> 0
  | Element _ -> 1
  | Concat (k, l) -> length(k) + length(l)
```

[1] Another recent integration of integers into the term rewriting framework is presented in [16]. The approach of [16] is incomparable to the approach of the present paper. On the one hand, [16] provides a more complete integration of integers since multiplication and division are supported. On the other hand, [16] does not consider collection data structures or context-sensitive rewriting.

```
let rec reverse x = match x with
  | Nil -> Nil
  | Element _ -> x
  | Concat (k, l) -> Concat (reverse l, reverse k)
```

Notice the use of equational attributes in both examples. These examples can be modeled using the following rewrite rules:

$$\begin{aligned}
\mathsf{length}(\mathsf{nil}) &\to 0\\
\mathsf{length}([n]) &\to 1\\
\mathsf{length}(k \mathbin{+\!\!+} l) &\to \mathsf{length}(k) + \mathsf{length}(l)\\
\mathsf{reverse}(\mathsf{nil}) &\to \mathsf{nil}\\
\mathsf{reverse}([n]) &\to [n]\\
\mathsf{reverse}(k \mathbin{+\!\!+} l)) &\to \mathsf{reverse}(l) \mathbin{+\!\!+} \mathsf{reverse}(k)
\end{aligned}$$

A suitable treatment of the equational attributes that closely follows the treatment in Maude and OCaml is discussed in Sect. 2. ◊

Even though CERSs are an expressive and elegant tool for modeling algorithms, they do not incorporate reduction strategies that are commonly used in declarative specification and programming languages such as Maude [9]. Context-sensitive rewriting [25,27] has been introduced as an operational restriction of term rewriting that can be used to model such reduction strategies (the close relationship between context-sensitive rewriting and Maude's `strat`-annotations has been investigated in [26]). Furthermore, context-sensitive rewriting allows to model lazy evaluation as used in functional programming languages such as Haskell (the relationship between lazy evaluation and context-sensitive rewriting has been investigated in [28]). In context-sensitive rewriting, a *replacement map* specifies the arguments where an evaluation may take place for each function symbol, and a reduction is only allowed at a position that is not forbidden by a function symbol occurring somewhere above it. The second contribution of this paper is to introduce context-sensitive rewriting for CERSs, thus combining the expressiveness of CERSs with increased flexibility for the reduction strategy.

Example 2. This example demonstrates the use of context-sensitive rewriting and integers in Maude.

```
fmod LAZY-LISTS is
  protecting INT .
  sorts List LazyList .
  op nil : -> List [ ctor ] .
  op cons : Int List -> List [ ctor ] .
  op lazycons : Int LazyList -> LazyList [ ctor strat (1) ] .
  op from : Int -> LazyList .
  op take : Int LazyList -> List .
  var M N : Int .
  var L : LazyList .
  eq from(N) = lazycons(N, from(N + 1)) .
  ceq take(N, L) = nil if N <= 0 .
  ceq take(N, lazycons(M, L)) = cons(M, take(N - 1, L)) if N > 0 .
endfm
```

Here, the `strat`-annotation specifies that only the first argument of `lazycons` may be reduced, i.e., the second argument of `lazycons` is "frozen". This example can be modeled using the following rewrite rules, where the constraints of the take-rules directly correspond to the `if`-conditions used in Maude:

$$\begin{aligned}\mathsf{from}(n) &\to \mathsf{lazycons}(n, \mathsf{from}(n+1))\\ \mathsf{take}(n, l) &\to \mathsf{nil}\ [\![n \le 0]\!]\\ \mathsf{take}(n, \mathsf{lazycons}(m, l)) &\to \mathsf{cons}(m, \mathsf{take}(n-1, l))\ [\![n > 0]\!]\end{aligned}$$

The semantics of the `strat`-annotation is modeled by a replacement map with $\mu(\mathsf{lazycons}) = \{1\}$. Notice that the use of CERSs makes it very easy to model many functional Maude modules occurring in practice. $\Diamond$

Termination also is a fundamental property of context-sensitive rewriting. As illustrated by Ex. 2, context-sensitive rewriting may result in a terminating rewrite relation where regular rewriting is not terminating. Thus, proving termination of context-sensitive rewriting is quite challenging.

For ordinary TRSs, there are two approaches to proving termination of context-sensitive rewriting. The first approach is to apply a syntactic transformation in such a way that termination of context-sensitive rewriting with a TRS is implied by (regular) termination of the TRS obtained by the transformation. For details on this approach, see [17,30]. While the application of these transformations allows the use of any method for proving termination of the transformed TRS, they often generate TRSs whose termination is hard to establish.

The second approach consists of the development of dedicated methods for proving termination of context-sensitive rewriting. Examples for adaptations of classical methods are context-sensitive recursive path orderings [8] and context-sensitive polynomial interpretations [29]. The main drawback of these adaptations is the limited power which is inherited from the classical methods. Adapting the more powerful dependency pair method [4] to context-sensitive TRSs has been a challenge. A first adaptation of the dependency pair method to context-sensitive TRSs has been presented in [2]. But this adaptation has severe disadvantages since it requires collapsing dependency pairs. An alternative adaptation of the dependency pair method to context-sensitive TRSs has recently been presented in [1]. This adaptation does not require collapsing dependency pairs and makes it much easier to adapt techniques developed within the ordinary dependency pair method to the context-sensitive case.

The third and main contribution of this paper is the development of a dependency pair method for context-sensitive rewriting with CERSs, taking [1] as a starting point. This adaptation is non-trivial since [1] is concerned with ordinary (syntactic) rewriting, whereas rewriting with CERSs is based on normalized equational rewriting that uses constructor equations and constructor rules. While the techniques presented in this paper are quite similar to the corresponding techniques in [1], their soundness proofs are more complex and cannot be presented due to space limitations. They can be found in the full version [14].

The techniques developed in this paper have been fully implemented in the termination prover AProVE [18]. The implementation has been successfully evaluated on a large collection of examples. This evaluation shows that the implementation succeeds in proving termination of many context-sensitive CERSs corresponding to functional Maude modules and OCaml programs.

After fixing terminology, Sect. 2 recalls and extends the CERSs introduced in [12]. In contrast to [12] which only supports natural numbers, it is now possible to consider built-in integers. Context-sensitive rewriting with CERSs is introduced in Sect. 3. The main technical result of this paper is presented in Sect. 4. By a non-trivial extension of [1], termination of context-sensitive rewriting with a CERS is reduced to showing absence of infinite chains of dependency pairs. Sect. 5 introduces several powerful termination techniques that can be applied in combination with dependency pairs. These techniques lift the most commonly used termination techniques introduced for CERSs in [12] to context-sensitive CERSs. An implementation of these techniques in AProVE [18] is discussed and evaluated in Sect. 6.

2 Constrained Equational Rewrite Systems

Familiarity with the notation and terminology of term rewriting is assumed, see [5] for an in-depth treatment. This paper uses many-sorted term rewriting over a set S of sorts. It is assumed in the following that all terms, substitutions, replacements, etc. are sort-correct. For a signature $\mathcal{F}$ and a disjoint set $\mathcal{V}$ of variables, the set of all terms over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The set of positions of a term t is denoted by $\mathcal{P}os(t)$, where Λ denotes the root position. The set of variables occurring in a term t is denoted by $\mathcal{V}(t)$, and $\mathcal{F}(t)$ denotes the set of function symbols occurring in t. This naturally extends to pairs of terms, sets of terms, etc. The root symbol of a term t is denoted by $\mathrm{root}(t)$.

A *context* over $\mathcal{F}$ is a term $C \in \mathcal{T}(\mathcal{F} \cup \bigcup_{s \in S}\{\Box_s\}, \mathcal{V})$. Here, $\Box_s : \to s$ is a fresh constant symbol of sort s, called *hole*. If the sort of a hole can be derived or is not important, then $\Box$ will be used to stand for any of the $\Box_s$. If C is a context with n holes and $t_1, \ldots, t_n$ are terms of the appropriate sorts, then $C[t_1, \ldots, t_n]$ is the result of replacing the occurrences of holes by $t_1, \ldots, t_n$ "from left to right". A *substitution* is a mapping from variables to terms, where the domain of the substitution may be infinite. The application of a substitution σ to a term t is written as $t\sigma$, using postfix notation.

A finite set $\mathcal{E} = \{u_1 \approx v_1, \ldots, u_n \approx v_n\}$ of equations induces a rewrite relation $\to_\mathcal{E}$ by letting $s \to_\mathcal{E} t$ iff there exist a position $p \in \mathcal{P}os(s)$ and a substitution σ such that $s|_p = u_i\sigma$ and $t = s[v_i\sigma]_p$ for some $u_i \approx v_i \in \mathcal{E}$. The reflexive-transitive-symmetric closure of $\to_\mathcal{E}$ is denoted by $\sim_\mathcal{E}$. If equations are used in only one direction, they are called *rules*. A *term rewrite system (TRS)* is a finite set $\mathcal{R} = \{l_1 \to r_1, \ldots, l_m \to r_m\}$ of rules. Equational rewriting uses both a set $\mathcal{E}$ of equations and a set $\mathcal{R}$ of rules. Intuitively, $\mathcal{E}$ is used to model "structural" properties, while $\mathcal{R}$ is used to model "simplifying" properties.

Definition 3 ($\mathcal{E}$-Extended Rewriting). *Let $\mathcal{R}$ be a TRS and let $\mathcal{E}$ be a set of equations. Then $s \to_{\mathcal{E}\backslash\mathcal{R}} t$ if there exist a rule $l \to r \in \mathcal{R}$, a position $p \in \mathcal{P}os(s)$, and a substitution σ such that (i) $s|_p \sim_\mathcal{E} l\sigma$, and (ii) $t = s[r\sigma]_p$.*

Writing $\overset{>\Lambda}{\sim}_{\mathcal{E}}$ and $\xrightarrow{>\Lambda}_{\mathcal{E}\backslash\mathcal{R}}$ denotes that all steps are applied below the root, and $\xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\backslash\mathcal{R}}$ denotes normalization with $\xrightarrow{>\Lambda}_{\mathcal{E}\backslash\mathcal{R}}$.

In order to allow for built-in numbers and collection data structures, [12] has introduced a new class of rewrite systems. Both built-in numbers and collection data structures are modeled using $\mathcal{E}$-extended rewriting. In order to model the set of integers, recall that $\mathbb{Z}$ is an Abelian group with unit $\mathsf{0}$ that is generated using the element $\mathsf{1}$. Integers can thus be modeled using the function symbols $\mathcal{F}_{\mathbb{Z}} = \{\mathsf{0} : \to \texttt{int},\ \mathsf{1} : \to \texttt{int},\ - : \texttt{int} \to \texttt{int},\ + : \texttt{int} \times \texttt{int} \to \texttt{int}\}$. Terms over $\mathcal{F}_{\mathbb{Z}}$ are written using a simplified notation, e.g., $x - \mathsf{2}$ instead of $x + ((-\mathsf{1}) + (-\mathsf{1}))$.

As is well-known, *equational completion* [23,6] generates the following rules $\mathcal{S}_{\mathbb{Z}}$ and equations $\mathcal{E}_{\mathbb{Z}}$ from the defining properties of Abelian groups:

$$\begin{array}{rlrl}
x + \mathsf{0} & \to x & x + (-x) & \to \mathsf{0} \\
- - x & \to x & (x + (-x)) + y & \to \mathsf{0} + y \\
-\mathsf{0} & \to \mathsf{0} & x + y & \approx y + x \\
-(x + y) & \to (-x) + (-y) & x + (y + z) & \approx (x + y) + z
\end{array}$$

Recall that equality w.r.t. the properties of Abelian groups is reduced to $\mathcal{E}_{\mathbb{Z}}$-equivalence of $\to_{\mathcal{E}_{\mathbb{Z}}\backslash\mathcal{S}_{\mathbb{Z}}}$-normal forms. This idea can be used for natural numbers with $\mathcal{F}_{\mathbb{N}} = \{\mathsf{0} : \to \texttt{nat},\ \mathsf{1} : \to \texttt{nat},\ + : \texttt{nat} \times \texttt{nat} \to \texttt{nat}\}$, $\mathcal{S}_{\mathbb{N}} = \{x + \mathsf{0} \to x\}$, and $\mathcal{E}_{\mathbb{N}} = \{x + y \approx y + x,\ x + (y + z) \approx (x + y) + z\}$ as well [12]. In the following, $\mathcal{N}um$ denotes one of $\mathbb{Z}$ or $\mathbb{N}$, and `num` denotes the sort `int` or `nat`, respectively.

Properties of the built-in numbers are modeled using the predicate symbols $\mathsf{P} = \{>,\ \geq,\ \simeq\}$. The rewrite rules that are used in order to specify the defined function symbols are equipped with constraints over these predicate symbols that guard when a rewrite step may be performed. An *atomic $\mathcal{N}um$-constraint* has the form $t_1\ P\ t_2$ for a predicate symbol $P \in \mathsf{P}$ and terms $t_1, t_2 \in \mathcal{T}(\mathcal{F}_{\mathcal{N}um}, \mathcal{V})$. The set of *$\mathcal{N}um$-constraints* is the closure of the set of atomic $\mathcal{N}um$-constraints under $\top$ (truth), $\neg$ (negation), and $\wedge$ (conjunction). The Boolean connectives $\vee$, $\Rightarrow$, and $\Leftrightarrow$ can be defined as usual. Also, $\mathcal{N}um$-constraints have the expected semantics. The main interest is in *$\mathcal{N}um$-satisfiability* (i.e., the constraint is true for some instantiation of its variables) and *$\mathcal{N}um$-validity* (i.e., the constraint is true for all instantiations of its variables). Both of these properties are decidable.

In order to extend $\mathcal{F}_{\mathcal{N}um}$ by collection data structures and defined functions, a finite signature $\mathcal{F}$ over the sort `num` and a new sort `univ` is used. The restriction to two sorts is not essential, but the techniques presented in the remainder of this paper only need to differentiate between terms of sort `num` and terms of any other sort. Collection data structures can be handled similarly to the built-in numbers by using equational completion on their defining properties [11,12], see Fig. 1. In the following, a combination of $\mathcal{N}um$ with (signature-disjoint) collection data structures $\mathcal{C}_1, \ldots, \mathcal{C}_n$ is considered. In order to do so, let $\mathcal{S} = \mathcal{S}_{\mathcal{N}um} \cup \bigcup_{i=1}^{n} \mathcal{S}_{\mathcal{C}_i}$ and $\mathcal{E} = \mathcal{E}_{\mathcal{N}um} \cup \bigcup_{i=1}^{n} \mathcal{E}_{\mathcal{C}_i}$.

Definition 4 (Constrained Rewrite Rules). *A* constrained rewrite rule *has the form $l \to r[\![\varphi]\!]$ for terms $l, r \in \mathcal{T}(\mathcal{F} \cup \mathcal{F}_{\mathcal{N}um}, \mathcal{V})$ and a $\mathcal{N}um$-constraint φ such that* $\mathrm{root}(l) \in \mathcal{F} - \mathcal{F}(\mathcal{E} \cup \mathcal{S})$ *and $\mathcal{V}(r) \cup \mathcal{V}(\varphi) \subseteq \mathcal{V}(l)$.*

	Constructors	$\mathcal{S}_\mathcal{C}$ and $\mathcal{E}_\mathcal{C}$
Compact Lists	$\mathsf{nil}, \mathsf{ins}$	$\mathsf{ins}(x, \mathsf{ins}(x, ys)) \to \mathsf{ins}(x, ys)$
Compact Lists	$\mathsf{nil}, [\cdot], +\!\!+$	$x +\!\!+ \mathsf{nil} \to x$ $\mathsf{nil} +\!\!+ y \to y$ $[x] +\!\!+ [x] \to [x]$ $x +\!\!+ (y +\!\!+ z) \approx (x +\!\!+ y) +\!\!+ z$
Multisets	$\emptyset, \mathsf{ins}$	$\mathsf{ins}(x, \mathsf{ins}(y, zs)) \approx \mathsf{ins}(y, \mathsf{ins}(x, zs))$
Multisets	$\emptyset, \{\cdot\}, \cup$	$x \cup \emptyset \to x$ $x \cup (y \cup z) \approx (x \cup y) \cup z$ $x \cup y \approx y \cup x$
Sets	$\emptyset, \mathsf{ins}$	$\mathsf{ins}(x, \mathsf{ins}(x, ys)) \to \mathsf{ins}(x, ys)$ $\mathsf{ins}(x, \mathsf{ins}(y, zs)) \approx \mathsf{ins}(y, \mathsf{ins}(x, zs))$
Sets	$\emptyset, \{\cdot\}, \cup$	$x \cup \emptyset \to x$ $x \cup x \to x$ $(x \cup x) \cup y \to x \cup y$ $x \cup (y \cup z) \approx (x \cup y) \cup z$ $x \cup y \approx y \cup x$

Fig. 1. Modeling collection data structures

In a constrained rewrite rule $l \to r[\![\top]\!]$, the constraint $\top$ is usually omitted. A finite set $\mathcal{R}$ of constrained rewrite rules and the sets $\mathcal{S}$ and $\mathcal{E}$ for modeling $\mathcal{N}um$ and collection data structures as given above are combined into a *constrained equational rewrite system (CERS)*[2] $(\mathcal{R}, \mathcal{S}, \mathcal{E})$.

The rewrite relation of a CERS is defined as follows [12]: First, the redex is normalized by $\to_{\mathcal{E}\backslash\mathcal{S}}$. Then, the redex is $\mathcal{E}$-matched to the left-hand side of a rewrite rule and it is checked whether the matching substitution makes the constraint of that rewrite rule $\mathcal{N}um$-valid. Notice that checking the instantiated constraint for validity requires the matching substitution to be $\mathcal{N}um$-based, i.e., all variables of sort `num` have to be mapped to terms from $\mathcal{T}(\mathcal{F}_{\mathcal{N}um}, \mathcal{V})$.

Definition 5 (Rewrite Relation of a CERS). *For a CERS $(\mathcal{R}, \mathcal{S}, \mathcal{E})$, let $s \xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R}} t$ iff there exist $l \to r[\![\varphi]\!] \in \mathcal{R}$, a position $p \in \mathcal{P}os(s)$, and a $\mathcal{N}um$-based substitution σ such that (i) $s|_p \xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\backslash\mathcal{S}} \circ \overset{>\Lambda}{\sim}_{\mathcal{E}} l\sigma$, (ii) $\varphi\sigma$ is $\mathcal{N}um$-valid, and (iii) $t = s[r\sigma]_p$.*

It is shown in [14] that $\xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R}}$ is decidable for the CERSs considered in this paper. The function symbols occurring at the root position of left-hand sides in $\mathcal{R}$ are of particular interest since they are the only function symbols that allow a reduction to take place. These are the *defined symbols* $\mathcal{D}(\mathcal{R})$.

Example 6. This example is closely related to Ex. 2, but instead of a list the function from now generates a set built using $\emptyset$ and ins as in Fig. 1. Consider the following rewrite rules:

[2] A more abstract definition of CERSs that allows for more general non-free data structures is given in [14]. The main requirement is that $\to_{\mathcal{E}\backslash\mathcal{S}}$ needs is convergent.

$$\begin{aligned}
\mathsf{from}(x) &\to \mathsf{ins}(x, \mathsf{from}(x+1)) \\
\mathsf{take}(0, xs) &\to \mathsf{nil} \\
\mathsf{take}(x, \mathsf{ins}(y, ys)) &\to \mathsf{cons}(y, \mathsf{take}(x-1, ys))[\![x > 0]\!] \\
\mathsf{pick}(\mathsf{ins}(x, xs)) &\to x \\
\mathsf{drop}(\mathsf{ins}(x, xs)) &\to xs
\end{aligned}$$

Notice that the term $\mathsf{take}(2, \mathsf{from}(0))$ admits an infinite reduction in which the from-rule is applied again and again. However, there also is a finite reduction of that term which results in the normal form $\mathsf{cons}(0, \mathsf{cons}(1, \mathsf{nil}))$. This reduction can be enforced using context-sensitive rewriting, cf. Ex. 8. $\Diamond$

3 Context-Sensitive Rewriting with CERSs

A context-sensitive rewriting strategy is given using a *replacement map* μ with $\mu(f) \subseteq \{1, \ldots, \text{arity}(f)\}$ for every function symbol $f \in \mathcal{F} \cup \mathcal{F}_{\mathcal{N}um}$. Replacement maps specify the argument positions of function symbols where reductions are allowed. If the replacement map restricts reductions in a certain argument position, then the whole subterm below that argument position may not be reduced. Formally, μ is used to define the set $\mathcal{P}os^{\mu}(t)$ of *active* positions of a term t. Here, a position is active if it can be reached from the root of the term by only descending into argument positions that are not restricted by the replacement map, i.e., $\mathcal{P}os^{\mu}(x) = \{\Lambda\}$ for $x \in \mathcal{V}$ and $\mathcal{P}os^{\mu}(f(t_1, \ldots, t_n)) = \{\Lambda\} \cup \{i.p \mid i \in \mu(f) \text{ and } p \in \mathcal{P}os^{\mu}(t_i)\}$. Dually, the set of *inactive positions* of t is defined as $\mathcal{P}os^{\neg\mu}(t) = \mathcal{P}os(t) - \mathcal{P}os^{\mu}(t)$. The context-sensitive rewrite relation of a CERS will be obtained by a small modification of Def. 5 such that the position where the reduction takes place has to be active, see Def. 7 below.

The concept of active positions can also be used to define active (and inactive) subterms of a given term. $t \trianglerighteq_{\mu} s$ denotes that s is an *active subterm* of t, i.e., $t|_p = s$ for an active position $p \in \mathcal{P}os^{\mu}(t)$. If $p \neq \Lambda$, then this is written $t \triangleright_{\mu} s$. Analogously, $t \triangleright_{\neg\mu} s$ means that s is an *inactive subterm* of t. The classification of active and inactive subterms can easily be extended to other notions as well to obtain the sets $\mathcal{V}^{\mu}(t)$ of variables occurring in active positions in t, $\mathcal{V}^{\neg\mu}(t)$ of variables occurring in inactive positions in t, etc.

Now a *context-sensitive constrained equational rewrite system (CS-CERS)* $(\mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$ combines a regular CERS with a replacement map. As already noticed in [15] for the AC-case, the permutative nature of the equations in $\mathcal{E}$ disallows some choices of μ since inactive subterms may otherwise become active subterms (or vice versa) by applying equations from $\mathcal{E}$. Therefore, μ needs to satisfy the following conditions:

$$\begin{array}{ll}
\mu(+) = \{1, 2\} & \mu(\mathsf{ins}) = \emptyset \text{ or } \mu(\mathsf{ins}) = \{1, 2\} \\
\mu(-) = \{1\} & \mu(\mathbin{+\!\!+}) = \mu(\cup) = \{1, 2\}
\end{array}$$

As mentioned above, the rewrite relation of a CS-CERS is obtained by a small modification of Def. 5 such that the position where the reduction takes place has to be active.

Definition 7 (Rewriting with a CS-CERS). *For a CS-CERS $(\mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$, let $s \xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu} t$ iff there exist $l \to r[\![\varphi]\!] \in \mathcal{R}$, an active position $p \in \mathcal{P}os^{\mu}(s)$, and a $\mathcal{N}um$-based substitution σ such that (i) $s|_p \xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\backslash\mathcal{S}} \circ \overset{>\Lambda}{\sim}_{\mathcal{E}} l\sigma$, (ii) $\varphi\sigma$ is $\mathcal{N}um$-valid, and (iii) $t = s[r\sigma]_p$.*

Example 8. The CERS from Ex. 6 becomes a CS-CERS by considering the replacement map μ with $\mu(\mathsf{ins}) = \emptyset$ and $\mu(f) = \{1, \ldots, \text{arity}(f)\}$ for all $f \neq \mathsf{ins}$. Then the reduction of the term $\mathsf{take}(2, \mathsf{from}(0))$ has the following form:

$$\begin{aligned}\mathsf{take}(2, \mathsf{from}(0)) &\xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu} \mathsf{take}(2, \mathsf{ins}(0, \mathsf{from}(1)))\\ &\xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu} \mathsf{cons}(0, \mathsf{take}(2-1, \mathsf{from}(1)))\\ &\xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu} \mathsf{cons}(0, \mathsf{cons}(1, \mathsf{take}(1-1, \mathsf{from}(2))))\\ &\xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu} \mathsf{cons}(0, \mathsf{cons}(1, \mathsf{nil}))\end{aligned}$$

Notice that an infinite reduction of this term from Ex. 6 is not possible since the recursive call in the rule $\mathsf{from}(x) \to \mathsf{ins}(x, \mathsf{from}(x+1))$ occurs in an inactive position. ◊

4 Dependency Pairs for Rewriting with CS-CERSs

Recall from [4] that dependency pairs are built from recursive calls to defined symbols occurring in right-hand sides of $\mathcal{R}$ since only these recursive calls may cause non-termination. As usual, a signature $\mathcal{F}^{\sharp}$ is introduced, containing the function symbol $f^{\sharp} : s_1 \times \ldots \times s_n \to \mathtt{top}$ for each function symbol $f : s_1 \times \ldots \times s_n \to s$ from $\mathcal{D}(\mathcal{R})$. Here, $\mathtt{top}$ is a fresh sort. For $t = f(t_1, \ldots, t_n)$, the term $f^{\sharp}(t_1, \ldots, t_n)$ is denoted by $t^{\sharp}$. A dependency pair generated from a rule $l \to r[\![\varphi]\!]$ has the shape $l^{\sharp} \to t^{\sharp}[\![\varphi]\!]$, where t is a subterm of r with $\text{root}(t) \in \mathcal{D}(\mathcal{R})$. The main theorem for CERSs [12] states that a CERS is terminating if it is not possible to construct infinite *chains* from the dependency pairs.

For context-sensitive rewriting, one might be tempted to restrict the generation of dependency pairs to recursive calls occurring in active positions since these are the only places where reductions may occur. As shown in [2] for ordinary TRSs, this results in an unsound method if rules have *migrating variables*, i.e., variables x with $r \trianglerighteq_{\mu} x$ but $l \ntrianglerighteq_{\mu} x$ for some rule $l \to r$. In Ex. 6 and 8, the variable x is migrating in the pick-rule and xs is migrating in the drop-rule. The reason that migrating variables require attention is that recursive calls occurring in inactive positions might be promoted to active positions if they are matched to a migrating variable of another rule. Thus, [2] introduces collapsing dependency pairs for such migrating variables, but this causes severe disadvantages that make it hard to extend methods for proving termination from ordinary rewriting to context-sensitive rewriting. While progress has been made [2,3,20], the resulting methods are quite weak in practice.

An alternative to the collapsing dependency pairs needed in [2] has recently been presented in [1]. The main observation of [1] is that only certain instantiations of the migrating variables need to be considered. A first, naive approach for this would be to consider only instantiations by *hidden terms*, which are terms with a defined root symbol occurring inactively in right-hand sides of rules.

Definition 9 (Hidden Term). *A term t is* hidden *for $(\mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$ iff* $\mathrm{root}(t) \in \mathcal{D}(\mathcal{R})$ *and there exists a rule $l \to r[\![\varphi]\!] \in \mathcal{R}$ such that $r \rhd_{\neg\mu} t$.*

In Ex. 8, the term $\mathsf{from}(x+1)$ is hidden since $\mathsf{ins}(x, \mathsf{from}(x+1)) \rhd_{\neg\mu} \mathsf{from}(x+1)$. As shown in [1] for ordinary TRSs, it does not suffice to consider only the hidden terms. Instead, it becomes necessary to consider certain contexts that may be built above a hidden term using the rewrite rules. Formally, this observation is captured using the notion of *hiding contexts*. The definition in this paper generalizes the one given in [1] by also considering $\mathcal{S}$ and $\mathcal{E}$.

Definition 10 (Hiding Contexts). *Given a CS-CERS $(\mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$, $f \in \mathcal{F} \cup \mathcal{F}_{\mathcal{N}um}$* hides position *$i$ iff $i \in \mu(f)$ and either $f \in \mathcal{F}(\mathcal{E} \cup \mathcal{S})$ or there exist a rule $l \to r[\![\varphi]\!] \in \mathcal{R}$ and a term $s = f(s_1, \ldots, s_i, \ldots, s_n)$ with $r \rhd_{\neg\mu} s$ and $s_i \unrhd_\mu x$ for an $x \in \mathcal{V}$ or $s_i \unrhd_\mu g(\ldots)$ with $g \in \mathcal{D}(\mathcal{R})$. A context C is* hiding *iff $C = \Box$ or $C = f(t_1, \ldots, t_{i-1}, C', t_{i+1}, \ldots, t_n)$ where f hides position i and C' is hiding.*

In Ex. 8, $+$ hides positions 1 and 2 and $-$ and from hide position 1. Notice that there are infinitely many hiding contexts, but that these hiding context have a regular shape. In order to represent all hiding contexts using only finitely many dependency pairs, fresh function symbols $\mathsf{U}_{\mathtt{num}}$ and $\mathsf{U}_{\mathtt{univ}}$ and *unhiding* dependency pairs are used. The purpose of these unhiding dependency pairs is to extract a hidden term from a hiding context surrounding it.

Definition 11 (Context-Sensitive Dependency Pairs). *Let $(\mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$ be a CS-CERS. The set of* context-sensitive dependency pairs *of $\mathcal{R}$ is defined as $\mathsf{DP}(\mathcal{R}, \mu) = \mathsf{DP}_\mathsf{o}(\mathcal{R}, \mu) \cup \mathsf{DP}_\mathsf{u}(\mathcal{R}, \mu)$ where*

$$\begin{aligned}
\mathsf{DP}_\mathsf{o}(\mathcal{R}, \mu) &= \{l^\sharp \to t^\sharp[\![\varphi]\!] \mid l \to r[\![\varphi]\!] \in \mathcal{R}, r \unrhd_\mu t, \mathrm{root}(t) \in \mathcal{D}(\mathcal{R})\} \\
\mathsf{DP}_\mathsf{u}(\mathcal{R}, \mu) &= \{l^\sharp \to \mathsf{U}_s(x)[\![\varphi]\!] \mid l \to r[\![\varphi]\!] \in \mathcal{R}, r \unrhd_\mu x, l \not\rhd_\mu x\} \\
&\cup \{\mathsf{U}_s(g(x_1, \ldots, x_i, \ldots, x_n)) \to \mathsf{U}_{s'}(x_i)[\![\top]\!] \mid g \textit{ hides position } i\} \\
&\cup \{\mathsf{U}_s(h) \to h^\sharp[\![\top]\!] \mid h \textit{ is a hidden term}\}
\end{aligned}$$

Here, $\mathsf{U}_{\mathtt{num}} : \mathtt{num} \to \mathtt{top}$ and $\mathsf{U}_{\mathtt{univ}} : \mathtt{univ} \to \mathtt{top}$ are fresh function symbols that are added to $\mathcal{F}^\sharp$ and s and s' are the appropriate sorts. Furthermore, $\mu(\mathsf{U}_{\mathtt{num}}) = \mu(\mathsf{U}_{\mathtt{univ}}) = \emptyset$ and $\mu(f^\sharp) = \mu(f)$ for all $f \in \mathcal{F}$.

Example 12. For Ex. 8, $\mathsf{DP}(\mathcal{R}, \mu)$ is as follows:

$$\mathsf{take}^\sharp(x, \mathsf{ins}(y, ys)) \to \mathsf{take}^\sharp(x - 1, ys)\ [\![x > 0]\!] \tag{1}$$

$$\mathsf{take}^\sharp(x, \mathsf{ins}(y, ys)) \to \mathsf{U}_{\mathtt{num}}(y)\ [\![x > 0]\!] \tag{2}$$

$$\mathsf{take}^\sharp(x, \mathsf{ins}(y, ys)) \to \mathsf{U}_{\mathtt{univ}}(ys)\ [\![x > 0]\!] \tag{3}$$

$$\mathsf{pick}^\sharp(\mathsf{ins}(x, xs)) \to \mathsf{U}_{\mathtt{num}}(x) \tag{4}$$

$$\mathsf{drop}^\sharp(\mathsf{ins}(x, xs)) \to \mathsf{U}_{\mathtt{univ}}(xs) \tag{5}$$

$$\mathsf{U}_{\mathtt{univ}}(\mathsf{from}(x+1)) \to \mathsf{from}^\sharp(x+1) \tag{6}$$

$$\mathsf{U}_{\mathtt{num}}(x+y) \to \mathsf{U}_{\mathtt{num}}(x) \tag{7}$$
$$\mathsf{U}_{\mathtt{num}}(x+y) \to \mathsf{U}_{\mathtt{num}}(y) \tag{8}$$
$$\mathsf{U}_{\mathtt{num}}(-x) \to \mathsf{U}_{\mathtt{num}}(x) \tag{9}$$
$$\mathsf{U}_{\mathtt{univ}}(\mathsf{from}(x)) \to \mathsf{U}_{\mathtt{num}}(x) \tag{10}$$

For this, recall the hidden terms and the hiding contexts from above. ♢

As usual in methods based on dependency pairs, context-sensitive dependency pairs can be used in order to build chains, and the goal is to show that $\xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu}$ is terminating if there are no infinite minimal chains.

Definition 13 ($(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$-Chains). *Let $(\mathcal{R},\mathcal{S},\mathcal{E},\mu)$ be a CS-CERS and let $\mathcal{P}$ be a set of dependency pairs. A (variable-renamed) sequence of dependency pairs $s_1 \to t_1[\![\varphi_1]\!], s_2 \to t_2[\![\varphi_2]\!], \ldots$ from $\mathcal{P}$ is a $(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$*-chain *iff there exists a $\mathcal{N}um$-based substitution σ such that $t_i\sigma \xrightarrow{\mathcal{S}}{}^{*}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu} \circ \xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\backslash\mathcal{S}} \circ \overset{>\Lambda}{\sim}_{\mathcal{E}} s_{i+1}\sigma$ and $\varphi_i\sigma$ is $\mathcal{N}um$-valid for all $i \geq 1$. The above $(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$-chain is* minimal *iff $t_i\sigma$ does not start an infinite $\xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu}$-reduction for all $i \geq 1$.*

Here, $\xrightarrow{\mathcal{S}}{}^{*}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu}$ corresponds to reductions occurring strictly below the root of $t_i\sigma$ and $\xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\backslash\mathcal{S}} \circ \overset{>\Lambda}{\sim}_{\mathcal{E}}$ corresponds to normalization and matching before applying $s_{i+1} \to t_{i+1}[\![\varphi_i]\!]$ at the root position. Notice that this definition of chains is essentially identical to the non-context-sensitive case in [12]. Proving the following result for CS-CERSs constitutes the main technical contribution of this paper. The proof requires several technical lemmas that handle the subtle interplay between $\mathcal{R}$, $\mathcal{S}$, $\mathcal{E}$, and μ.[3] It can be found in the full version [14].

Theorem 14. *Let $(\mathcal{R},\mathcal{S},\mathcal{E},\mu)$ be a CS-CERS. Then $\xrightarrow{\mathcal{S}}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu}$ is terminating if there are no infinite minimal $(\mathsf{DP}(\mathcal{R},\mu),\mathcal{R},\mathcal{S},\mathcal{E},\mu)$-chains.*

In the next section, several techniques for showing absence of infinite chains are presented. These techniques are given in the form of *CS-DP processors* that operate on *CS-DP problems* in the spirit of [19]. Here, a CS-DP problem has the form $(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$, where $\mathcal{P}$ is a finite set of dependency pairs and $(\mathcal{R},\mathcal{S},\mathcal{E},\mu)$ is a CS-CERS. A CS-DP processor is a function that takes a CS-DP problem as input and returns a finite set of CS-DP problems as output. A CS-DP processor Proc is *sound* iff for all CS-DP problems $(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$ with an infinite minimal $(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$-chain there exists a CS-DP problem $(\mathcal{P}',\mathcal{R}',\mathcal{S}',\mathcal{E}',\mu') \in \mathsf{Proc}(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$ with an infinite minimal $(\mathcal{P}',\mathcal{R}',\mathcal{S}',\mathcal{E}',\mu')$-chain. For a termination proof of the CS-CERS $(\mathcal{R},\mathcal{S},\mathcal{E},\mu)$, sound CS-DP processors are applied recursively to the initial CS-DP problem $(\mathsf{DP}(\mathcal{R},\mu),\mathcal{R},\mathcal{S},\mathcal{E},\mu)$. If all resulting CS-DP problems have been transformed into $\emptyset$, then termination has been shown.

[3] Amongst others, it needs to be shown that an application of rules from $\mathcal{S}$ and equations from $\mathcal{E}$ transforms a hiding context into another hiding context. For example, the hiding context $\mathsf{from}(\Box + \mathsf{1})$ is transformed into the hiding context $\mathsf{from}(\mathsf{1} + \Box)$ by the equation $x + y \approx y + x$.

5 CS-DP Processors

This section introduces several sound CS-DP processors. Most of these processors are similar to corresponding processors for the non-context-sensitive case [12].

5.1 Dependency Graphs

Like the corresponding DP processor from [12], the CS-DP processor introduced in this section decomposes a CS-DP problem into several independent CS-DP problems by determining which dependency pairs from $\mathcal{P}$ may follow each other in a $(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$-chain. The processor relies on the notion of *(estimated) dependency graphs*, which has initially been introduced for ordinary TRSs [4]. Here, the estimation from [12] is adaptated using an approach similar to [2,1].

Definition 15 (Estimated Context-Sensitive Dependency Graphs). *For a CS-DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$, the nodes of the* estimated $(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$-dependency graph $\mathsf{EDG}(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$ *are the dependency pairs in $\mathcal{P}$ and there is an arc from $s_1 \to t_1[\![\varphi_1]\!]$ to $s_2 \to t_2[\![\varphi_2]\!]$ iff there is a substitution σ such that $\textsc{cap}_\mu(t_1)\sigma \overset{>\Lambda}{\longrightarrow}{}^{!}_{\mathcal{E}\backslash\mathcal{S}} \circ \overset{>\Lambda}{\sim}_{\mathcal{E}} s_2\sigma$ and $\varphi_1\sigma$, $\varphi_2\sigma$ are $\mathcal{N}um$-valid. $\textsc{cap}_\mu$ is given by*

1. *for $x \in \mathcal{V}$, $\textsc{cap}_\mu(x) = x$ if $\mathrm{sort}(x) = \mathtt{num}$ and $\textsc{cap}_\mu(x) = y$ otherwise,*
2. *$\textsc{cap}_\mu(f(t_1, \ldots, t_n)) = f(t'_1, \ldots, t'_n)$ if $f \notin \mathcal{D}(\mathcal{R})$, where $t'_i = t_i$ if $i \notin \mu(f)$ and $t'_i = \textsc{cap}_\mu(t_i)$ if $i \in \mu(f)$, and*
3. *$\textsc{cap}_\mu(f(t_1, \ldots, t_n)) = y$ if $f \in \mathcal{D}(\mathcal{R})$.*

Here, y is the next variable in an infinite list $y_1, y_2, \ldots$ of fresh variables.

Incomplete methods to implement this estimation are given in [10].

Theorem 16 (CS-DP Processor Using Dependency Graphs). *The CS-DP processor with $\mathsf{Proc}(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu) = \{(\mathcal{P}_1, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu), \ldots, (\mathcal{P}_n, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)\}$, where $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are the strongly connected components (SCCs) of the estimated dependency graph $\mathsf{EDG}(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$, is sound.*

Example 17. For the dependency pairs from Ex. 12, the following estimated dependency graph $\mathsf{EDG}(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$ is obtained:

$$\begin{tikzcd}
 & & (1) \arrow[dl] \arrow[dr] \arrow[loop below] & & (6) & \\
(4) \arrow[dr] & (2) \arrow[d] & & (3) \arrow[ur] \arrow[dr] & & (5) \arrow[ul] \arrow[dl] \\
 & (7),(8),(9) \arrow[loop left] & & & (10) \arrow[lll] &
\end{tikzcd}$$

Here, the nodes for (7)–(9) have been combined since they have "identical" incoming and outgoing arcs. This estimated dependency graph contains two SCCs, and, according to Thm. 16, the following CS-DP problems are obtained:

$$(\{(1)\}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu) \tag{11}$$

$$(\{(7), (8), (9)\}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu) \tag{12}$$

These CS-DP problem can now be handled independently of each other. ♢

5.2 Subterm Criterion

The subterm criterion for ordinary TRSs [21] is a relatively simple technique which is nonetheless surprisingly powerful. The technique works particularly well for functions that are defined using primitive recursion. The subterm criterion applies a *projection* which collapses a term $f^\sharp(t_1, \ldots, t_n)$ to one of the t_i.

Definition 18 (Projections). *A* projection *is a mapping π that assigns to every $f^\sharp \in \mathcal{F}^\sharp$ with* $\text{arity}(f^\sharp) = n$ *an i with $1 \leq i \leq n$. The mapping that assigns to every term $f^\sharp(t_1, \ldots, t_n)$ the term $t_{\pi(f^\sharp)}$ is also denoted by π.*

After applying a projection, the subterm relation modulo $\mathcal{E}$ is used. For CS-CERSs, this relation needs to take the replacement map into account by only considering subterms in active positions. This is similar to [2].

Definition 19 ($\mathcal{E}$-μ-Subterms). *Let $(\mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$ be a CS-CERS and let s, t be terms. Then t is a* strict $\mathcal{E}$-μ-subterm of *s, written $s \rhd_{\mathcal{E},\mu} t$, iff $s \sim_{\mathcal{E}} \circ \rhd_\mu \circ \sim_{\mathcal{E}} t$. The term t is an* $\mathcal{E}$-μ-subterm of *s, written $s \unrhd_{\mathcal{E},\mu} t$, iff $s \rhd_{\mathcal{E},\mu} t$ or $s \sim_{\mathcal{E}} t$.*

The subterm criterion is now implemented by the following CS-DP processor. Notice that the sets $\mathcal{R}$ and $\mathcal{S}$ do not need to be considered when operating on the CS-DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$.

Theorem 20 (CS-DP Processor Using the Subterm Criterion). *For a projection π, let* Proc *be a CS-DP processor with* $\mathsf{Proc}(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu) =$

- $\{(\mathcal{P} - \mathcal{P}', \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)\}$, *if $\mathcal{P}' \subseteq \mathcal{P}$ such that*
  - $\pi(s) \rhd_{\mathcal{E},\mu} \pi(t)$ *for all* $s \to t[\![\varphi]\!] \in \mathcal{P}'$, *and*
  - $\pi(s) \unrhd_{\mathcal{E},\mu} \pi(t)$ *for all* $s \to t[\![\varphi]\!] \in \mathcal{P} - \mathcal{P}'$.
- $(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$, *otherwise.*

Then Proc *is sound.*

Example 21. Recall the CS-DP problem (12) from Ex. 17, consisting of the dependency pair (7)–(9). Using $\pi(\mathsf{U}_{\mathsf{num}}) = 1$, this CS-DP problem can easily be handled since all dependency pairs are removed from it. ◊

5.3 Reduction Pairs

As usual in methods based on dependency pairs, well-founded relations on terms may be used in order to remove dependency pairs from CS-DP problems. Often, *reduction pairs* [24] are used for this purpose, and they can immediately be applied for CS-CERSs as well. If the CS-CERS uses built-in natural numbers, then $\mathcal{PA}$-reduction pairs [12] may be used. Here, it is shown that a special class of polynomial interpretations is applicable if integers are built-in.[4]

[4] It is also possible to develop a general framework of $\mathbb{Z}$-*reduction pairs* [10].

A *$\mathbb{Z}$-polynomial interpretation $\mathcal{P}ol$* fixes a constant $c_{\mathcal{P}ol} \in \mathbb{Z}$ and maps

1. the symbols in $\mathcal{F}_{\mathbb{Z}}$ to polynomials over $\mathbb{Z}$ in the natural way, i.e., $\mathcal{P}ol(\mathsf{0}) = 0$, $\mathcal{P}ol(\mathsf{1}) = 1$, $\mathcal{P}ol(-) = -x_1$ and $\mathcal{P}ol(+) = x_1 + x_2$,
2. the symbols in $\mathcal{F}$ to polynomials over $\mathbb{N}$ such that $\mathcal{P}ol(f) \in \mathbb{N}[x_1, \ldots, x_n]$ if arity$(f) = n$, and
3. the symbols in $\mathcal{F}^\sharp$ to polynomials over $\mathbb{Z}$ such that $\mathcal{P}ol(f^\sharp) \in \mathbb{Z}[x_1, \ldots, x_n]$ if arity$(f^\sharp) = n$ and $\mathcal{P}ol(f^\sharp)$ is weakly increasing in all x_i where the i^{th} argument of $f^\sharp$ has sort `univ`.

Terms are mapped to polynomials by defining $[x]_{\mathcal{P}ol} = x$ for variables $x \in \mathcal{V}$ and $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = \mathcal{P}ol(f)([t_1]_{\mathcal{P}ol}, \ldots, [t_n]_{\mathcal{P}ol})$.

Definition 22 ($\succ_{\mathcal{P}ol}$, $\gtrsim_{\mathcal{P}ol}$, and $\sim_{\mathcal{P}ol}$ for $\mathbb{Z}$-Polynomial Interpretations). *Let $\mathcal{P}ol$ be a $\mathbb{Z}$-polynomial interpretation. Then $s \succ_{\mathcal{P}ol} t$ iff $[s\sigma]_{\mathcal{P}ol} \geq c_{\mathcal{P}ol}$ and $[s\sigma]_{\mathcal{P}ol} > [t\sigma]_{\mathcal{P}ol}$ for all ground substitutions $\sigma : \mathcal{V}(s) \cup \mathcal{V}(t) \to \mathcal{T}(\mathcal{F} \cup \mathcal{F}_{\mathbb{Z}})$. Analogously, $s \gtrsim_{\mathcal{P}ol} t$ iff $[s\sigma]_{\mathcal{P}ol} \geq [t\sigma]_{\mathcal{P}ol}$ for all ground substitutions $\sigma : \mathcal{V}(s) \cup \mathcal{V}(t) \to \mathcal{T}(\mathcal{F} \cup \mathcal{F}_{\mathbb{Z}})$ and $s \sim_{\mathcal{P}ol} t$ iff $[s\sigma]_{\mathcal{P}ol} = [t\sigma]_{\mathcal{P}ol}$ for all ground substitutions $\sigma : \mathcal{V}(s) \cup \mathcal{V}(t) \to \mathcal{T}(\mathcal{F} \cup \mathcal{F}_{\mathbb{Z}})$.*

For constrained terms, it suffices to consider all substitutions σ that make the constraint $\mathbb{Z}$-valid. This is similar to the $\mathcal{PA}$-reduction pairs of [12].

Definition 23 ($\succ_{\mathcal{P}ol}$ and $\gtrsim_{\mathcal{P}ol}$ on Constrained Terms). *Let $\mathcal{P}ol$ be a $\mathbb{Z}$-polynomial interpretation, let s, t be terms and let φ be a $\mathbb{Z}$-constraint. Then $s[\![\varphi]\!] \gtrsim_{\mathcal{P}ol} t[\![\varphi]\!]$ iff $s\sigma \gtrsim_{\mathcal{P}ol} t\sigma$ for all $\mathbb{Z}$-based substitutions σ such that $\varphi\sigma$ is $\mathbb{Z}$-valid. Similarly, $s[\![\varphi]\!] \succ_{\mathcal{P}ol} t[\![\varphi]\!]$ iff $s\sigma \succ_{\mathcal{P}ol} t\sigma$ for all $\mathbb{Z}$-based substitutions σ such that $\varphi\sigma$ is $\mathbb{Z}$-valid.*

Thus, $s[\![\varphi]\!] \succ_{\mathcal{P}ol} t[\![\varphi]\!]$ if the following formulas are true in the integers (here, $x_1, \ldots, x_n$ are the variables occurring in $[s]_{\mathcal{P}ol}$ or $[t]_{\mathcal{P}ol}$):

$$\begin{aligned} &\forall x_1, \ldots, x_n.\ \varphi \Rightarrow [s]_{\mathcal{P}ol} \geq c_{\mathcal{P}ol} \\ &\forall x_1, \ldots, x_n.\ \varphi \Rightarrow [s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol} \end{aligned}$$

Since $\mathcal{P}ol(-)$ is not monotonic in its argument, it becomes necessary to impose restrictions on the CS-DP problem under which $\mathbb{Z}$-polynomial interpretations may be applied. More precisely, it has to be ensured that no reduction with $\overset{\mathcal{S}}{\to}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu}$ takes place below an occurrence of $-$. The easiest way to ensure this is as follows: If all arguments of right-hand sides from $\mathcal{P}$ are terms in $\mathcal{T}(\mathcal{F}_{\mathbb{Z}}, \mathcal{V})$, then no reduction with $\overset{\mathcal{S}}{\to}_{\mathcal{N}um\|\mathcal{E}\backslash\mathcal{R},\mu}$ can take place between instantiated dependency pairs in a chain since chains are built using $\mathbb{Z}$-based substitutions. Additional sufficient conditions and refined techniques are presented in [14].

Theorem 24 (CS-DP Processor Using $\mathbb{Z}$-Polynomial Interpretations). *Let* Proc *be a CS-DP processor with* $\mathsf{Proc}(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu) =$

- $\{(\mathcal{P} - \mathcal{P}', \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)\}$, *if all arguments of right-hand sides of $\mathcal{P}$ are terms from $\mathcal{T}(\mathcal{F}_{\mathbb{Z}}, \mathcal{V})$, $\mathcal{P}ol$ is a $\mathbb{Z}$-polynomial interpretation, $\mathcal{P}' \subseteq \mathcal{P}$, and*

  - $s[\![\varphi]\!] \succ_{\mathcal{P}ol} t[\![\varphi]\!]$ *for all* $s \to t[\![\varphi]\!] \in \mathcal{P}'$
  - $s[\![\varphi]\!] \succsim_{\mathcal{P}ol} t[\![\varphi]\!]$ *for all* $s \to t[\![\varphi]\!] \in \mathcal{P} - \mathcal{P}'$
- $\{(\mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)\}$, *otherwise.*

Then Proc *is sound.*

If $\mathcal{P}$ contains right-hand sides with arguments that are not from $\mathcal{T}(\mathcal{F}_{\mathbb{Z}}, \mathcal{V})$, then it might be possible to use a *non-collapsing argument filter* [24] for the function symbols $f^\sharp \in \mathcal{F}^\sharp$ that ensures that this condition is satisfied afterwards.

Example 25. Recall the CS-DP problem (11) from Ex. 17, consisting of the dependency pair (1). Using a non-collapsing argument filtering that only retains the first argument of $\mathsf{take}^\sharp$, this dependency pair is transformed into

$$\mathsf{take}^\sharp(x) \to \mathsf{take}^\sharp(x - 1)\ [\![x > \mathsf{0}]\!]$$

Now Thm. 24 can be applied and using $c_{\mathcal{P}ol} = 0$ and $\mathcal{P}ol(\mathsf{take}^\sharp) = x_1$ concludes the termination proof of the running example since

$$\forall x.\ x > 0 \Rightarrow x \geq 0$$
$$\forall x.\ x > 0 \Rightarrow x > x - 1$$

are true in the integers. ♢

6 Evaluation and Conclusions

This paper has presented a generalization of the constrained equational rewrite systems (CERSs) introduced in [12]. Then, context-sensitive rewriting strategies for these generalized CERSs have been investigated. The main interest has been in the automated termination analysis for such context-sensitive CERSs. For this, a dependency pair framework for CS-CERSs has been developed, taking the recent method of [1] for ordinary context-sensitive TRSs as a starting point. Then, many of the DP processors developed for non-context-sensitive rewriting in [12] have been adapted to the context-sensitive case.

The techniques presented in this paper have been fully implemented in the termination prover AProVE [18], resulting in AProVE-CERS.

While most of the implementation is relatively straightforward, the computation of the estimated dependency graph is non-trivial since, given s and t, it needs to be checked whether there exists a σ such that $s\sigma \xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\backslash\mathcal{S}} \circ \overset{>\Lambda}{\sim}_{\mathcal{E}} t\sigma$. Notice that this is a generalization of $\mathcal{E}$-unifiability that stems from the normalization process used in CERSs. If $\mathcal{E}$- or $\mathcal{E} \cup \mathcal{S}$-unifiability is decidable, then the above problem can be approximated by these. Otherwise, syntactic unifiability can be used as a (weak) approximation. Details on this can be found in [10].

An automatic generation of $\mathbb{Z}$-polynomial interpretations is non-trivial as well since the constraints of the rewrite rules need to be utilized. This is done by first using the constraints in order to derive upper and/or lower bounds on variables. These upper and/or lower bounds are then used in combination with absolute positiveness [22] in order to automatically generate a suitable (concrete) $\mathbb{Z}$-polynomial interpretations from a parametric $\mathbb{Z}$-polynomial interpretation (i.e., a $\mathbb{Z}$-polynomial interpretation where the coefficients are parameters that need to be instantiated). The approach is discussed in detail in [13].

In order to evaluate the effectiveness of the approach on "typical" algorithms, the implementation has been evaluated on a collection of 150 (both context-sensitive and non-context-sensitive) examples. Most of these examples stem from the *Termination Problem Data Base*, suitably adapted to make use of built-in integers and/or collection data structures. The majority of examples correspond to functional programs as written in OCaml. Additionally, the collection contains several examples corresponding to functional Maude modules taken from [9] that operate on sets or multisets. The collection furthermore contains more than 40 examples that were obtained by encoding programs from the literature on termination proving of imperative programs into CERSs, see [13].

With a time limit of 60 seconds for each example, AProVE-CERS succeeds in proving termination of 140 (93.3%) of the examples, taking an average time of 2.15 seconds for each example.[5] An empirical comparison with AProVE-Integer based on the methods presented in [16] has been conducted on a subset of 80 examples where the methods of [16] are applicable (i.e., examples that use neither context-sensitive strategies nor collection data structures).[6] Out of these 80 examples, AProVE-CERS succeeds on 73, while AProVE-Integer succeeds on 72. There are examples that can only be handled by AProVE-CERS but not by AProVE-Integer, and vice versa. On examples that can be handled by both AProVE-CERS and AProVE-Integer, the system AProVE-CERS that is based on the present paper is much faster than AProVE-Integer, on average by a factor of three (in the most extreme case, AProVE-CERS succeeds in 0.1s while AProVE-Integer needs 52.7s in order to prove termination). The detailed empirical evaluation, including all termination proofs generated by AProVE-CERS and AProVE-Integer, is available at `http://www.cs.unm.edu/~spf/tdps/`.

References

1. Alarcón, B., Emmes, F., Fuhs, C., Giesl, J., Gutiérrez, R., Lucas, S., Schneider-Kamp, P., Thiemann, R.: Improving context-sensitive dependency pairs. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 636–651. Springer, Heidelberg (2008)
2. Alarcón, B., Gutiérrez, R., Lucas, S.: Context-sensitive dependency pairs. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 297–308. Springer, Heidelberg (2006)
3. Alarcón, B., Gutiérrez, R., Lucas, S.: Improving the context-sensitive dependency graph. ENTCS 188, 91–103 (2007)
4. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236(1-2), 133–178 (2000)

[5] Five of the 150 examples produce a timeout after 60 seconds. Without these examples, the average time on the remaining 145 examples is 0.16 seconds.

[6] Notice, in particular, that AProVE-Integer is not applicable to the examples from [9].

5. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
6. Bachmair, L., Dershowitz, N.: Completion for rewriting modulo a congruence. TCS 67(2-3), 173–201 (1989)
7. Blanqui, F., Hardin, T., Weis, P.: On the implementation of construction functions for non-free concrete data types. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 95–109. Springer, Heidelberg (2007)
8. Borralleras, C., Lucas, S., Rubio, A.: Recursive path orderings can be context-sensitive. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 314–331. Springer, Heidelberg (2002)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
10. Falke, S.: Term Rewriting with Built-In Numbers and Collection Data Structures. PhD thesis, University of New Mexico, Albuquerque, NM, USA (2009)
11. Falke, S., Kapur, D.: Dependency pairs for rewriting with non-free constructors. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 426–442. Springer, Heidelberg (2007)
12. Falke, S., Kapur, D.: Dependency pairs for rewriting with built-in numbers and semantic data structures. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 94–109. Springer, Heidelberg (2008)
13. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. LNCS, vol. 5663, pp. 277–293. Springer, Heidelberg (2009)
14. Falke, S., Kapur, D.: Termination of context-sensitive rewriting with built-in numbers and collection data structures. Technical Report TR-CS-2009-01 (2009)
15. Ferreira, M.C.F., Ribeiro, A.L.: Context-sensitive AC-rewriting. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 286–300. Springer, Heidelberg (1999)
16. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving termination of integer term rewriting. In: Treinen, R. (ed.) Rewriting Techniques and Applications. LNCS, vol. 5595, pp. 32–47. Springer, Heidelberg (2009)
17. Giesl, J., Middeldorp, A.: Transformation techniques for context-sensitive rewrite systems. JFP 14(4), 379–427 (2004)
18. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
19. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
20. Gutiérrez, R., Lucas, S., Urbain, X.: Usable rules for context-sensitive rewrite systems. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 126–141. Springer, Heidelberg (2008)
21. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. IC 205(4), 474–511 (2007)
22. Hong, H., Jakuš, D.: Testing positiveness of polynomials. JAR 21, 23–38 (1998)
23. Jouannaud, J.-P., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM J. Comput. 15(4), 1155–1194 (1986)
24. Kusakari, K., Nakamura, M., Toyama, Y.: Argument filtering transformation. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 47–61. Springer, Heidelberg (1999)

25. Lucas, S.: Context-sensitive computations in functional and functional logic programs. JFLP 1998(1) (1998)
26. Lucas, S.: Termination of rewriting with strategy annotations. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 669–684. Springer, Heidelberg (2001)
27. Lucas, S.: Context-sensitive rewriting strategies. IC 178(1), 294–343 (2002)
28. Lucas, S.: Lazy rewriting and context-sensitive rewriting. ENTCS 64, 234–254 (2002)
29. Lucas, S.: Polynomials for proving termination of context-sensitive rewriting. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 318–332. Springer, Heidelberg (2004)
30. Lucas, S.: Proving termination of context-sensitive rewriting by transformation. IC 204(12), 1782–1846 (2006)

Semantic Labelling for Proving Termination of Combinatory Reduction Systems

Makoto Hamana

Department of Computer Science, Gunma University, Japan
hamana@cs.gunma-u.ac.jp

Abstract. We give a novel transformation method for proving termination of higher-order rewrite rules in Klop's format called Combinatory Reduction System (CRS). The format CRS essentially covers the usual pure higher-order functional programs such as Haskell. Our method called higher-order semantic labelling is an extension of a method known in the theory of term rewriting. This attaches semantics of the arguments to each function symbol. We systematically define the labelling by using the complete algebraic semantics of CRS, Σ-monoids. We also examine the power of higher-order semantic labelling by several examples. This includes an interesting example from the viewpoint of functional programming.

1 Introduction

Rewrite rules appear everywhere in computer science. In programming language theory, we use often transformation of states, expressions, terms, or programs given by some form of rewrite rules. Functional programs such as Haskell can also be regarded as rewrite rules. When reasoning with such rewrite rules, termination is one of the most important property, because it is necessary for decidable equality checking. This topic has been extensively investigated in the field of term rewriting [BN98, Ter03].

In this paper, we deal with higher-order rewrite rules in Klop's format called combinatory reduction systems (CRSs) [Klo80, KOR93]. The format CRS is known as one of the most early detailed formulation of higher-order rewriting systems (i.e. rewriting systems having the feature of *variable binding* and meta-level substitutions) in the theory of term rewriting. A CRS is a set of rewrite rules on second-order terms. We give a method to prove termination, meaning *strong normalisation*, of a CRS by a translation called *higher-order semantic labelling*. This is an extension of a method for first-order term rewriting systems (TRSs) [Zan95].

Weakness of existing syntactic methods. Higher-order extensions of term rewriting systems [Ter03] are known as several formats: major representatives are CRSs, Higher-order Rewrite Systems [Nip91], and Inductive Data Type Systems [BJO02]. There exist several termination criteria: higher-order recursive path order (HORPO) [JR07], the General Schema [BJO02, Bla00], hereditary monotone functional interpretation [Pol94], binding algebra interpretation [Ham05]. Recently improvements of HORPO/General Schema are actively investigated [BR01, Raa01, JR06]. A recent survey on this field can be found in [BJR08].

For CRSs, the General Schema is a decidable syntactic criteria of termination [Bla00]. The idea of the General Schema is to control the arguments of the right-hand side recursive calls in a rewrite rule by checking that they are smaller than the left-hand sides ones in the strict subterm order extended in a multiset or lexicographic manner.

The General Schema criteria is effective for rewrite rules defined *structural recursively* on term structures. However, there are many realistic rewrite rules which do not fit into this scheme. We use often rewrite rules defined *non-structural recursively* on term structures. The General Schema is *not* applicable to prove termination of such rewrite rules. What does this mean? This implies that rather than syntactic structures, *semantic structures* should be more explicit in many situations, and they can be a "hint" of complete termination proofs. Our approach to give termination proofs in this paper does not aim to be fully automatic. Finding appropriate semantics of rewrite rules automatically is hard in general. But in most cases, *one has the intended semantics for rewrite rules in one's mind that matches the intended application* (if not, why one could write such rewrite rules?) Hence, making such semantics explicit is merely a matter of formulation. Notice that our method is not fully semantical either. We combine both syntactic and semantical information. Below, we give two examples to illustrate this situation.

Example 1 (The prefix sum of a list). Consider the following CRS $\mathcal{P}$ for computing the prefix sum of a list, i.e., the list with the sum of all prefixes of a given list using the higher-order function map (taken from [BR01]).

$$\begin{aligned} \mathsf{map}(a.\textsc{f}[a], \mathsf{nil}) &\to \mathsf{nil} \\ \mathsf{map}(a.\textsc{f}[a], \mathrm{x} : \mathrm{xs}) &\to \textsc{f}[\mathrm{x}] : \mathsf{map}(a.\textsc{f}[a], \mathrm{xs}) \\ \mathsf{ps}(\mathsf{nil}) &\to \mathsf{nil} \\ \mathsf{ps}(\mathrm{x} : \mathrm{xs}) &\to \mathrm{x} : \mathsf{ps}(\mathsf{map}(a.\mathrm{x} + a, \mathrm{xs})) \end{aligned}$$

We want to prove termination of the CRS $\mathcal{P}$. Unfortunately, the CRS $\mathcal{P}$ does not follow the General Schema, hence the exiting syntactic method is not powerful enough. This is because the argument of ps in the right-hand side of the last rule is *not* a subterm of the argument of ps in the left-hand side. However, we know that the map function does not change the length of a list, thus a shorter list than $x : xs$ is always used in the recursive call of ps. To prove termination of ps, this "semantic" information (rather than only syntactical structures) should be effectively used.

Higher-order semantic labelling developed in this paper solves this problem. It is a method to reflect such information in rewrite rules. In this case, we use the "length" of a list for ps as the semantics. Higher-order semantic labelling transforms the original CRS $\mathcal{P}$ to the following labelled CRS:

$$\begin{aligned} \mathsf{ps}_0(\mathsf{nil}) &\to \mathsf{nil} \\ \mathsf{ps}_{i+1}(\mathrm{x} : \mathrm{xs}) &\to \mathrm{x} : \mathsf{ps}_i(\mathsf{map}(a.\mathrm{x} + a, \mathrm{xs})) \end{aligned}$$

where $i \in \mathbb{N}$. This i denotes the "semantics" of argument ps, i.e., the length of a list. This transformation to attach semantics to function symbols is systematically defined in the structure of Σ-*monoids*, which is abstract algebraic structures of higher-order terms. Then, this labelled CRS successfully follows the General Schema with the precedence

$\mathsf{ps}_i > \mathsf{ps}_j > \mathsf{map} > :$ for $i > j \in \mathbb{N}$. This ordering is used to compare two term structures in a recursive manner. If two root function symbols can be compared by the ordering, thier arguments need not to be comparied. Hence, the order $\mathsf{ps}_{i+1} > \mathsf{ps}_i$ effectively solve the above mentioned problem on the mismatch between the term size and semantical size. Our main theorem (Thm. 10) of higher-order semantic labelling is that if the labelled CRS is terminating, then the original CRS is terminating. Hence, we can conclude termination of the CRS $\mathcal{P}$.

Example 2 (Haskell's rewrite rules). Glasgow Haskell Compiler (GHC) has a pragma called "rewrite rules" [JTH01] for an optimization purpose. GHC applies rewrite rules to the source program wherever it can. The following is an example that composes two `map`s in a program.

```
{-# RULES
  "map/map" forall f g xs.  map f (map g xs) = map (f.g) xs
#-}
```

Rewriting by this Haskell's rewrite rule are expected to be terminating at the compile time. But GHC makes no attempt to ensure that the rule is terminating because of complications of the combination of the rewrite rule and compiler's optimization rules [JTH01]. But ideally we should ensure termination of rewrite rules. To consider this problem in a formal setting, we model this Haskell's rewrite rule as the following CRS's rewrite rule:

$$\mathsf{map}(\textsc{f}, \mathsf{map}(\textsc{g}, \textsc{xs})) \to \mathsf{map}(a.\textsc{f}[\textsc{g}[a]], \textsc{xs})$$

A difficulty is that this rule is not defined structural recursively on a data structure. Moreover, the first argument of map in the right-hand side is bigger than ones in the left-hand side. Hence, this does not follow the General Schema.

Higher-order semantic labelling again solves this problem. We use "the number of maps" in the second argument of map as the semantics. Higher-order semantic labelling transforms the original rule to the following labelled rules:

$$\begin{array}{lll} \mathsf{map}_{i+1}(\textsc{f}, \mathsf{map}_i(\textsc{g}, \textsc{xs})) & \to \mathsf{map}_i(a.\textsc{f}[\textsc{g}[a]], \textsc{xs}) & \text{for all } i \in \mathbb{N} \\ \mathsf{map}_i(\textsc{f}, \textsc{xs}) & \to \mathsf{map}_j(\textsc{f}, \textsc{xs}) & \text{for all } i > j \in \mathbb{N} \end{array}$$

Each label denotes the number of maps in the second argument. The General Schema succeeds in showing termination of the rules with the precedence $\mathsf{map}_i > \mathsf{map}_j$ for all $i > j \in \mathbb{N}$. Hence by our main theorem, we conclude termination of the original map's rule.

These two examples use seemingly simple semantics, i.e., "the length of lists" and "the number of maps". But to compute them actually needs a sophisticated semantical account because the rewrite rules involve higher-order functions. Moreover, semantics need not to be numbers or "sizes". Arbitrary (higher-order) algebraic structures (e.g. λ-terms, domains, categories, etc.) can be semantics of a CRS. In other words, the semantic information for labels cannot obtained by just syntactic counting of symbols. In this paper, using the complete algebraic semantics Σ-monoid, we systematically give higher-order semantic labelling for CRSs.

Contribution. The contribution of this paper is summarised as follows.

(i) Theoretical contribution.
  - We generalised semantics labelling for TRSs [Zan95] to higher-order semantic labelling for CRSs in the framework of Σ-monoids. This also showed that Σ-monoids was certainly the right structure as the semantics of CRSs.
  - We showed that semantic labelled meta-terms form a Σ-monoid.
  - We identified the commutativity of the labelling operation with the substitutions appearing in formulation of CRSs is an essential property to establish semantic labelling.

(ii) Practical contribution. We demonstrate higher-order semantic labelling by several examples for which the General Schema alone fails.

Background. Semantic labelling on higher-order terms has been defined for Inductive Datatype Systems [Ham07]. The present paper much simplifies the labelling method to deal with CRSs. We also aim to apply it to examples taken from functional programming. The semantics used in this paper is based on the algebraic semantics of CRS Σ-monoids. The notion of Σ-monoids was introduced by Fiore, Plotkin and Turi [FPT99], then a higher-order abstract syntax for free Σ-monoids was developed by the author [Ham04]. The algebraic semantics for CRSs [Ham05] was an application of this Σ-monoid structure. The outline of semantic labelling for CRSs (without proofs) was presented at 13th International Conference on Logic for Programming Artificial Intelligence Reasoning (LPAR'06) as a short paper.

How to read the paper. Theories on term rewriting usually avoid the use of semantics as much as possible. In contrast to it, we rely on the semantics of higher-order terms and rewriting. The semantics structure Σ-monoid is a natural extension of the first-order universal algebra to the second-order setting by shifting the base category from **Set** to a presheaf category [FPT99]. It is systematically defined in the framework of categorical universal algebra. Why categorical notions are needed is to make definitions and discussions on higher-order rewriting mathematically simple, manageable and systematic. The seemingly "elementary" extension of first-order semantic structures to higher-order setting by hand (within ordinary set-theoretic setting) makes definitions and theories quite complex because of the combinations of ordinary first-order structures and higher-order structures. Category theory prevents this explosion by giving a right abstraction for algebraic and higher-order terms. Hence, this paper assumes basic knowledge of category theory for reading the development of the semantic labelling method, such as functor categories, monoidal categories, monoids and algebras (e.g. [Mac71] Chap. II, VII).

Future work. As a future work, we plan to make our method to be more accessible for users of proof assitants and dependently-typed programming languages. One of the most expected area that seriously needs termination proofs is proof assitants. Finding appropriate semantics fully automatic is impossible, but one's intended semantics might be directly mechanised within a proof assitant such as Coq or Agda. Combining it with syntactic methods will greatly reduce efforts to give full termination proofs in proof assitants. Hence, giving a convenient library and recipes for higher-order semantic labelling in a proof assitant

Organisation. This paper is organised as follows. We first review the definition of CRSs in Section 2 and the semantics of CRSs in Section 3. We give higher-order semantic labelling of CRSs in Section 4. In Section 5, we give the quasi-model version of higher-order semantic labelling and show several examples of termination proof using our method. All omitted proofs are given in Appendix.

2 Combinatory Reduction Systems

CRS. We review the definition of CRSs. We use the definition of the standard reference [KOR93] of CRSs with a slight modification of syntax used in [DR98]: $-.-$ and $-[-]$ instead of ordinary ones $[-]-$ and $-(-)$ in [KOR93].

Assume a signature Σ of function symbols f^l with arity, metavariables z^l with arity (in both cases the superscript $l \in \mathbb{N}$ is the arity).

(i) CRS *terms* have the form $t ::= x \mid x.t \mid f^l(t_1, \ldots, t_l)$. These forms are respectively called *variables*, *abstractions*, and *function terms*.

(ii) CRS *meta-terms* extend CRS terms to $t ::= x \mid x.t \mid f^l(t_1, \ldots, t_l) \mid \mathrm{z}^l[t_1, \ldots, t_l]$. The last form is called a *meta-application*.

(iii) A *valuation* θ is a mapping that assigns to n-ary metavariable z an n-ary *substitute* (a meta-level lambda notation, cf. [KOR93]) $\theta : \mathrm{z} \longmapsto \underline{\lambda}(x_1, \ldots, x_n).t$ where t is a term. Any valuation is extended to a function on meta-terms:

$$\begin{aligned} \theta(x) &= x & \theta(f(t_1, \ldots, t_l)) &= f(\theta(t_1), \ldots, \theta(t_l)) \\ \theta(x.t) &= x.\theta(t) & \theta(\mathrm{z}[t_1, \ldots, t_l]) &= \theta(\mathrm{z})\,(\theta(t_1), \ldots, \theta(t_l)) \end{aligned} \tag{1}$$

Note that the right-hand side of the equation (1) uses an application at the meta-level to the substitute. The valuation is *safe* if there are no two substitutes $\theta(\mathrm{z})$ and $\theta(\mathrm{z}')$ such that $\theta(\mathrm{z})$ contains a free variable x which appears also bound in $\theta(\mathrm{z}')$.

(iv) CRS *rules*, written $l \to r$, consist of two meta-terms l and r with the following additional restrictions:

(iv-a) l and r are closed (w.r.t. variables) meta-terms,

(iv-b) l must be a "pattern", i.e. a function term where all meta-applications have the form $\mathrm{z}[x_1, \ldots, x_n]$ with distinct variables x_i,

(iv-c) r can only contain meta-applications with meta-variables occurring in the left-hand side.

The rewrite rule $l \to r$ is *safe for* θ , if for all z in l and r, the substitute $\theta(\mathrm{z})$ does not have a free variable x occurring in an abstraction $x.-$ of l and r. A set of rewrite rules under the signature Σ is called a CRS and denoted by $(\Sigma, \mathcal{R})$ or simply $\mathcal{R}$.

(v) The CRS *rewrite relation* $\to_{\mathcal{R}}$ is generated by context and safe valuation closure of a given CRS $\mathcal{R}$:

$$\frac{l \to r \in \mathcal{R}}{\theta(l) \to_{\mathcal{R}} \theta(r)}\ \text{safe}\ \theta \qquad \frac{s \to_{\mathcal{R}} t}{x.s \to_{\mathcal{R}} x.t} \qquad \frac{s \to_{\mathcal{R}} t}{f(\ldots, s, \ldots) \to_{\mathcal{R}} f(\ldots, t, \ldots)}$$

where $l \to r$ must be safe for the safe valuation θ. The third rule means rewriting at the i-th argument of f. We say that $\mathcal{R}$ is *terminating* if $\to_{\mathcal{R}}$ is well-founded.

Structural CRSs. In this paper, we treat CRSs using (meta-)terms built from binding signatures, which we call *structural CRSs* (cf. Aczel's contraction schemes [Acz78]). A *binding signature* Σ consists of a set Σ of function symbols with an arity function $a : \Sigma \to \mathbb{N}^*$, where $\mathbb{N}^*$ denotes the set of all finite sequences of natural numbers. A function symbol of *binding arity* $\langle n_1, \ldots, n_l \rangle$, denoted by $f : \langle n_1, \ldots, n_l \rangle$, has l arguments and binds n_i variables in the i-th argument ($1 \le i \le l$). For a formal treatment of named variables modulo α-equivalence in CRSs, we assume the method of de Bruijn levels [dB72] for the naming convention of variables (N.B. not for metavariables) in CRSs. We also use the convention that $n \in \mathbb{N}$ denotes the set $\{1, \ldots, n\}$ (n is possibly 0). Under the method of de Bruijn levels, this n means the set of variables from 1 to n. *Structural meta-terms* are of the form $t ::= x \mid f(x_1 \cdots x_{i_1}.t_1, \ldots, x_1 \cdots x_{i_l}.t_l) \mid \mathrm{z}^l[t_1, \ldots, t_l]$ satisfying the restriction generated by the inference system given blow. Fix an $\mathbb{N}$-indexed set Z of metavariables defined by $Z(l) \triangleq \{\mathrm{z} \mid \mathrm{z} \text{ has arity } l\}$. A meta-term t is *structural* if $n \vdash t$ is derived from the following rules for some $n \in \mathbb{N}$.

$$\frac{x \in n}{n \vdash x} \qquad \frac{f : \langle i_1, \ldots, i_l \rangle \in \Sigma \quad n{+}i_1 \vdash t_1 \ \cdots \ n{+}i_l \vdash t_l}{n \vdash f(\, n{+}1 \ldots n{+}i_1.t_1,\ \ldots,\ n{+}1 \ldots n{+}i_l.t_l\,)}$$

$$\frac{\mathrm{z} \in Z(l) \quad n \vdash t_1 \ \cdots \ n \vdash t_l}{n \vdash \mathrm{z}[t_1, \ldots, t_l]}$$

By using these rules, we obtain meta-terms in the method of de Bruijn levels. A rewrite rule $1.\cdots n.l \to 1.\cdots n.r$ is called structural if l and r are structural, i.e. $n \vdash l$ and $n \vdash l$. A CRS is structural if all rules are structural. A valuation θ is structural if for any mapping by $\theta : \mathrm{z} \mapsto \underline{\lambda}(x_1, \ldots, x_n).t$, t is a structural term and all variables in t are included in $x_1, \ldots, x_n$. We may use the notation $Z|n \vdash s \to t$ for a rule or a rewrite step if metavariables and variables in s and t are included in Z and n respectively. We may also simply write $Z \vdash s \to t$ or $n \vdash s \to t$ if the other part is not important.

3 Semantics of CRSs

3.1 Binding Algebras

The semantics of CRS is given by the notion of binding algebras and Σ-monoids. What are Σ-monoids? A Σ-monoid is an algebra equipped with *substitution operation* on (semantics of) terms. This substitution operation is called *multiplication*, typically denoted by β in this paper. Why this is a multiplication is that the substitution operation satisfies the monoid law (imagine compositions of substitutions with the identity substitution) in an abstract setting.

We review the notion of binding algebras and Σ-monoids. For detail, see [FPT99]. Let $\mathbb{F}$ be the category which has finite cardinals $n = \{1, \ldots, n\}$ (n is possibly 0) as objects, and all functions between them as arrows. This is the category of object variables by the method of de Bruijn levels (i.e. natural numbers) and their renamings. We use the functor category $\mathbf{Set}^{\mathbb{F}}$. An object A of $\mathbf{Set}^{\mathbb{F}}$ is often called a *presheaf*. Subscripts may be used to denote parameters. The functor $\delta : \mathbf{Set}^{\mathbb{F}} \to \mathbf{Set}^{\mathbb{F}}$ for "index extension" is defined by $(\delta L)(n) = L(n+1)$ for $L \in \mathbf{Set}^{\mathbb{F}}$. To a binding signature Σ, we associate the *signature functor* $\Sigma : \mathbf{Set}^{\mathbb{F}} \to \mathbf{Set}^{\mathbb{F}}$ given by $\Sigma A = \coprod_{f:\langle n_1,\ldots,n_l\rangle \in \Sigma} \prod_{1 \le i \le l} \delta^{n_i} A$. A Σ-*algebra* is a pair (A, α) consisting of a presheaf $A \in \mathbf{Set}^{\mathbb{F}}$ called a *carrier* and a map ([ ] denotes a copair of coproducts) $\alpha = [f_A]_{f \in \Sigma} : \Sigma A \longrightarrow A$ called an *algebra structure*, where f_A is an *operation* $f_A : \delta^{n_1} A \times \ldots \times \delta^{n_l} A \longrightarrow A$ defined for each function symbol $f : \langle n_1, \ldots, n_l \rangle \in \Sigma$. The "presheaf of variables" $\mathrm{V} \in \mathbf{Set}^{\mathbb{F}}$ is defined by $\mathrm{V}(n) = n$, $\mathrm{V}(\rho) = \rho\ (\rho : m \to n$ in $\mathbb{F})$. For presheaves A and B, $(A \bullet B)(n) \triangleq (\coprod_{m \in \mathbb{N}} A(m) \times B(n)^m)/\sim$ where $\sim$ is the equivalence relation generated by $(t; u_{\rho 1}, \ldots, u_{\rho m}) \sim (A(\rho)(t); u_1, \ldots, u_l)$ for $\rho : m \to l$ in $\mathbb{F}$. Then, $(\mathbf{Set}^{\mathbb{F}}, \bullet, \mathrm{V})$ forms a monoidal category [Mac71], where the "substitution" monoidal product is defined as follows. An element of $A(m) \times B(n)^m$ is denoted by $(t; u_1, \ldots, u_m)$ where $t \in A(m)$ and $u_1, \ldots, u_m \in B(m)$. A representative of an equivalence class in $A \bullet B(n)$ is also denoted by this notation. Let Σ be a signature functor with strength st defined by a binding signature. A Σ-*monoid* $M = (M, \alpha, \eta, \mu)$ consists of a *monoid* $(M, \eta : \mathrm{V} \to M, \mu : M \bullet M \to M)$ in the monoidal category $(\mathbf{Set}^{\mathbb{F}}, \bullet, \mathrm{V})$ with a Σ algebra structure $\alpha : \Sigma M \to M$ satisfying $\mu \circ (\alpha \bullet \mathrm{id}_M) = \alpha \circ \Sigma\mu \circ \mathrm{st}$. A Σ-*monoid morphism* $M \longrightarrow M'$ is a morphism in $\mathbf{Set}^{\mathbb{F}}$ which is both Σ-algebra homomorphism and monoid morphism.

3.2 Algebra of Meta-terms

Let Z be an arbitrary $\mathbb{N}$-indexed set of metavariables (cf. Sec. 2). The *presheaf* $M_\Sigma Z$ *of meta-terms* is defined by $M_\Sigma Z(n) = \{t \mid n \vdash t\}$. We abbreviate $n{+}1, \ldots, n{+}k.t$ to $\overrightarrow{n{+}k}.t$. For every $f : \langle i_1, \ldots, i_l \rangle \in \Sigma$, we define the map $f_T : \delta^{i_1} M_\Sigma Z \times \cdots \times \delta^{i_l} M_\Sigma Z \longrightarrow M_\Sigma Z$ in $\mathbf{Set}^{\mathbb{F}}$ by $(t_1, \ldots, t_l) \longmapsto f(\overrightarrow{n{+}i_1}.t_1, \ldots, \overrightarrow{n{+}i_l}.t_l)$. The *multiplication* $\beta : M_\Sigma Z \bullet M_\Sigma Z \longrightarrow M_\Sigma Z$ is a map in $\mathbf{Set}^{\mathbb{F}}$ that performs a substitution of variables defined inductively as follows.

$$\beta(n)(i;\, \vec{t}) = t_i \qquad \beta(n)(\mathrm{z}[s_1, \ldots, s_l];\, \vec{t}) = \mathrm{z}[\beta(n)(s_1; \vec{t}), \ldots, \beta(n)(s_l; \vec{t})]$$

$$\begin{aligned}\beta(n)(f(\overrightarrow{m{+}i_1}.s_1, \ldots, \overrightarrow{m{+}i_l}.s_l);\, \vec{t}) = f(&\overrightarrow{m{+}i_1}.\beta(m{+}i_1)(s_1;\, \mathsf{up}_{i_1}(\vec{t}), m{+}1, \ldots, m{+}i_1), \ldots \\ &\overrightarrow{m{+}i_l}.\beta(m{+}i_l)(s_l;\, \mathsf{up}_{i_l}(\vec{t}), m{+}1, \ldots, m{+}i_l)\end{aligned}$$

where $f : \langle i_1, \ldots, i_l \rangle \in \Sigma$ and $\vec{t}$ denotes $t_1, \ldots, t_m$, and the weakening map from $M_\Sigma Z(m)$ to $M_\Sigma Z(m + i)$ is defined by $\mathsf{up}_i \triangleq M_\Sigma Z(\mathrm{id}_m + \mathsf{w}_i)$ where $\mathsf{w}_i : 0 \to i$. Then, the structural meta-terms $(M_\Sigma Z, [f_T]_{f \in \Sigma}, \nu, \beta)$ is a *free* Σ-*monoid* over a presheaf $\hat{Z}$, where $\nu : \mathrm{V} \longrightarrow M_\Sigma Z$ in $\mathbf{Set}^{\mathbb{F}}$ is defined by $x \longmapsto x$ and $\hat{Z}(n) = \coprod_{k \in \mathbb{N}} \mathbb{F}(k, n) \times Z(k)$ [Ham04]. Hereafter, given $\mathbb{N}$-indexed set Z, we abuse the notation to use Z to denote its presheaf version $\hat{Z} \in \mathbf{Set}^{\mathbb{F}}$ in an assignment.

Definition 3. We call an *assignment* a morphism $\phi : Z \longrightarrow A$ of $\mathbf{Set}^{\mathbb{F}}$ whose target A has a Σ-monoid structure (A, α, η, μ). By freeness, an assignment $\phi : Z \longrightarrow A$ is extended to the Σ-*monoid morphism* $\phi^* : M_\Sigma Z \longrightarrow A$ defined by

$$\begin{aligned}\phi^*_n(x) &= \eta_n(x) \qquad (x \in n)\\ \phi^*_n(f(\overrightarrow{n{+}i_1}.t_1, \ldots, \overrightarrow{n{+}i_l}.t_l)) &= f_A(\phi^*_{n+i_1}(t_1), \ldots, \phi^*_{n+i_l}(t_l))\\ \phi^*_n(\mathrm{z}[t_1, \ldots, t_l]) &= \mu_n(\, \phi_l(\mathrm{z});\, \phi^*_n(t_1), \ldots \phi^*_n(t_l)\,)\end{aligned}$$

where $f : \langle i_1, \ldots, i_l \rangle \in \Sigma$.

When the $\mathbb{N}$-indexed set of metavariables $Z = 0$ (empty set), $M_\Sigma 0$ is the presheaf of all structural terms (written as $T_\Sigma \mathrm{V}$ in [Ham05]). Moreover, $M_\Sigma 0$ forms the initial Σ-monoid [FPT99, Ham04]. An assignment $\theta : Z \longrightarrow M_\Sigma 0$ gives a structural valuation, and $\theta^* : M_\Sigma Z \longrightarrow M_\Sigma 0$ gives its "homomorphic" extension on meta-terms. We also call a *valuation* an assignment $\theta : Z \longrightarrow M_\Sigma 0$.

3.3 Algebraic Semantics of Rewriting

Henceforth, in this paper we consider structural CRSs only. So we say "a CRS" for a structural CRS.

The notions of models and quasi-models for CRSs are defined as follows. For a presheaf A, we write $\geq_A$ for a family of preorders $\{\geq_{A(n)}\}_{n\in\mathbb{N}}$, where $\geq_{A(n)}$ is a preorder on a set $A(n)$ for each $n \in \mathbb{N}$. Let $(A_1, \geq_{A_1}), \ldots, (A_l, \geq_{A_l}), (B, \geq_B)$ be presheaves equipped with preorders. A map $f : A_1 \times \cdots \times A_l \longrightarrow B$ in $\mathbf{Set}^{\mathbb{F}}$ is *weakly monotone* if all $n \in \mathbb{N}$, all $a_1, b_1 \in A_1(n), \ldots, a_l, b_l \in A_l(n)$ with $a_k \geq_{A(n)} b_k$ for some k and $a_j = b_j$ for all $j \neq k$, then $f(n)(a_1, \ldots, a_l) \geq_{B(n)} f(n)(b_1, \ldots, b_l)$. A *weakly monotone* $\mathrm{V}+\Sigma$*-algebra* $(A, \geq_A)$ is a $\mathrm{V}+\Sigma$-algebra $A = (A, [\nu, [f_A]_{f\in\Sigma}])$, where $\nu : \mathrm{V} \longrightarrow A$, equipped with preorders $\{\geq_{A(n)}\}_{n\in\mathbb{N}}$, such that every operation f_A is weakly monotone. Let A be a $\mathrm{V}+\Sigma$-algebra. A *term-generated assignment* $\phi : Z \longrightarrow A$ is a morphism of $\mathbf{Set}^{\mathbb{F}}$ that is expressed as the composite $Z \xrightarrow{\theta} M_\Sigma 0 \xrightarrow{!_A} A$ for some valuation θ, where $!_A$ is the unique $\mathrm{V}+\Sigma$-algebra homomorphism from the initial $\mathrm{V}+\Sigma$-algebra $M_\Sigma 0$. A $\mathrm{V}+\Sigma$-algebra A *satisfies* a rewrite rule $Z \vdash \vec{n}.l \to \vec{n}.r$ if $\phi^*(n)(l) = \phi^*(n)(r)$ for all term-generated assignments $\phi : Z \longrightarrow A$. A *model* A for a CRS $(\Sigma, \mathcal{R})$ is a $\mathrm{V}+\Sigma$-algebra A that satisfies all rules in the weakening closure $\mathcal{R}^\circ$ (cf. [Ham05]). A weakly monotone $\mathrm{V}+\Sigma$-algebra $(A, \geq_A)$ *satisfies* a rewrite rule $Z \vdash \vec{n}.l \to \vec{n}.r$ if $\phi^*(n)(l) \geq_{A(n)} \phi^*(n)(r)$ for all term-generated assignments $\phi : Z \longrightarrow A$. A *quasi-model* A for $(\Sigma, \mathcal{R})$ is a weakly monotone $\mathrm{V}+\Sigma$-algebra A that satisfies all rules in the weakening closure $\mathcal{R}^\circ$. An important fact is that any Σ-monoid (M, α, ν, μ) gives a $\mathrm{V}+\Sigma$-algebra $(M, [\nu, \alpha])$. Thus in this paper, we will basically work with Σ-monoids rather than V+Σ-algebras, which gives uniform semantic treatment of algebras with substitutions.

4 Higher-Order Semantic Labelling

We are now ready to give our semantic labelling for CRSs. We give an abstract formulation along the idea of initial algebra semantics in the framework of Σ-monoids.

4.1 Semantic Labelling for Meta-terms

Henceforth, we assume that Z is an $\mathbb{N}$-indexed set of metavariables, Σ is a binding signature and M is a Σ-monoid. We introduce labelling of functions symbols: choose for every $f \in \Sigma$ a corresponding non-empty set S_f of labels, called *semantic label set*. The binding signature $\overline{\Sigma}$ for labelled function symbols is defined by

$$\overline{\Sigma} = \{f_p \mid f \in \Sigma,\ p \in S_f\}$$

where the binding arity of f_p is defined to be the binding arity of f. A function symbol is labelled if S_f contains more than one element. For unlabelled f, the set S_f containing only one element can be left implicit; in that case we will often write f instead of f_p.

Choose for $f : \langle i_1, \dots, i_l \rangle \in \Sigma$, a *semantic label map* that is a morphism of $\mathbf{Set}^{\mathbb{F}}$ defined by

$$\langle\!\langle - \rangle\!\rangle^f : \delta^{i_1} M \times \cdots \times \delta^{i_l} M \longrightarrow K_{S_f}.$$

where $K_{S_f} \in \mathbf{Set}^{\mathbb{F}}$ is the constant presheaf defined by $K_{S_f}(n) = S_f$. If it is clear from the context, the superscript of $\langle\!\langle - \rangle\!\rangle^f$ will be omitted. The semantic label map was originally called a projection, denoted by π_f in [Zan95]. Then, as in the case of ordinary signature, we define $M_{\overline{\Sigma}} Z$ by the presheaf of all meta-terms generated by the labelled signature $\overline{\Sigma}$.

Definition 4 (Labelling map). Let $\phi : Z \longrightarrow M$ be an assignment. The *labelling map* $\phi^{\mathsf{L}} : M_\Sigma Z \longrightarrow M_{\overline{\Sigma}} Z$ is a morphism of $\mathbf{Set}^{\mathbb{F}}$ defined by

$$\phi^{\mathsf{L}}_n : M_\Sigma Z_n \longrightarrow M_{\overline{\Sigma}} Z_n$$

$$\phi^{\mathsf{L}}_n(x) = x \qquad \phi^{\mathsf{L}}_n(\mathrm{z}[\vec{t}]) = \mathrm{z}[\phi^{\mathsf{L}}_n \vec{t}]$$

$$\phi^{\mathsf{L}}_n(f(n+\vec{i_1}.t_1, \dots, n+\vec{i_l}.t_l)) = f_{\langle\!\langle \phi^*_{n+i_1}(t_1), \dots, \phi^*_{n+i_l}(t_l) \rangle\!\rangle^f_n}(n+\vec{i_1}.\phi^{\mathsf{L}}_{n+i_1} t_1, \dots, n+\vec{i_l}.\phi^{\mathsf{L}}_{n+i_l} t_l)$$

We state the following characterisation that clarifies what is the mathematical structure of semantic labelled meta-terms.

Theorem 5. *For each assignment $\phi : Z \longrightarrow M$, $(M_{\overline{\Sigma}} Z, [f_\phi]_{f \in \Sigma}, \nu_\phi, \beta_\phi)$ is a Σ-monoid.*

Corollary 6. *For each assignment $\phi : Z \to M$, the labelling map $\phi^{\mathsf{L}} : M_\Sigma Z \to M_{\overline{\Sigma}} Z$ is the unique Σ-monoid morphism $(M_\Sigma Z, [f_T]_{f \in \Sigma}, \nu, \beta) \to (M_{\overline{\Sigma}} Z, [f_\phi]_{f \in \Sigma}, \nu_\phi, \beta_\phi)$.*

Proof. Let $i_\phi : Z \to M_{\overline{\Sigma}} Z$ be the assignment into the Σ-monoid $(M_{\overline{\Sigma}} Z, [f_\phi]_{f \in \Sigma}, \nu_\phi, \beta_\phi)$ defined by $\mathrm{z} \mapsto \mathrm{z}$. It is clear that $i^*_\phi = \phi^{\mathsf{L}}$ by just comparing the definitions of ϕ^{L} and the Σ-monoid extension $(-)^*$. Hence ϕ^{L} gives a Σ-monoid morphism. □

Below we describes the Σ-monoid structure on $M_{\overline{\Sigma}} Z$ mentioned above for each assignment $\phi : Z \longrightarrow M$. Let $|-|$ be the function that erases all labels in a labeled meta-term for the ordinary signature Σ.

Unit. $\nu_\phi : \mathrm{V} \to M_{\overline{\Sigma}} Z$ is defined by $x \mapsto x$.

Operations. For $f : \langle i_1, \dots, i_l \rangle \in \Sigma$, the corresponding operation $f_\phi : \delta^{i_1} M_{\overline{\Sigma}} Z \times \cdots \times \delta^{i_l} M_{\overline{\Sigma}} Z \longrightarrow M_{\overline{\Sigma}} Z$ is defined by

$$f_\phi(n)(s_1, \dots, s_l) = f_{\langle\!\langle \phi^*_{n+i_1}(|s_1|), \dots, \phi^*_{n+i_l}(|s_l|) \rangle\!\rangle_n}(n + i_1.s_1, \dots, n + i_l.s_l).$$

Multiplication. $\beta_\phi : M_{\overline{\Sigma}} Z \bullet M_{\overline{\Sigma}} Z \longrightarrow M_{\overline{\Sigma}} Z$ is defined by

$$\beta_\phi(n)(x; \vec{t}) = t_x$$

$$\beta_\phi(n)(\mathrm{z}[s_1, \dots, s_l]; \vec{t}) = \mathrm{z}[\beta_\phi(n)(s_1; \vec{t}), \dots, \beta_\phi(n)(s_l; \vec{t})]$$

$$\beta_\phi(n)(f_q(m+\vec{i_1}.s_1, \dots, m+\vec{i_l}.s_l); \vec{t})$$

$$= \begin{cases} f_p(m+\vec{i_1}.\beta_\phi(m+i_1)(s_1;\ \mathsf{up}_{m+i_1}(\vec{t}), m+1, \dots, m+i_1), \dots) & \text{if } m+1 > n \\ f_p(n+\vec{i_1}.\beta_\phi(n+i_1)(s_1;\ \mathsf{up}_{n+i_1}(\vec{t}), n+1, \dots, n+i_1), \dots) & \text{if } m+1 \le n \end{cases}$$

where $p = \langle\!\langle \phi^*(n)|\beta_\phi(n+i_1)(s_1;\ \mathsf{up}_{i_1}(\vec{t}), n+1, \ldots, n+i_1)|, \ldots, \phi^*(n)|\beta_\phi(n+i_l)(s_l;\ \mathsf{up}_{i_l}(\vec{t}), n+1, \ldots, n+i_l)|\rangle\!\rangle_n$. For the third clause, we assume that m is the length of $\vec{t}$, and I is the maximum of $i_1, \ldots, i_l$, Note that the length of "$\mathsf{up}_{i_1}(\vec{t}), n{+}1, \ldots, n{+}i_1$" is $m + i_1$, and it renames $m + k$ by $n + k$ to make bound variables sense.

Laws. To check that $M_{\overline{\Sigma}}Z$ satisfies the monoid law is straightforward induction on meta-terms. To check the Σ-monoid law $\beta_\phi \circ ([f_\phi]_{f\in\Sigma} \bullet \mathrm{id}) = [f_\phi]_{f\in\Sigma} \circ \Sigma\beta_\phi \circ \mathrm{st}$, we instantiate this at $n \in \mathbb{F}$ and chase an element, this eventually becomes the equality

$$\beta_\phi(n)(f_r(m{+}\vec{i_1}.s_1, \ldots, m{+}\vec{i_l}.s_l);\ \vec{t}) = f_p(m{+}\vec{i_1}.\beta_\phi(m{+}i_1)(s_1;\ \mathsf{up}_{i_1}(\vec{t}), m{+}1, \ldots, m{+}i_1), \ldots$$
$$m{+}\vec{i_l}.\beta_\phi(m{+}i_l)(s_l;\ \mathsf{up}_{i_l}(\vec{t}), m{+}1, \ldots, m{+}i_l)$$

where $r = \langle\!\langle \phi^*_{n+i_1}(|s_1|), \ldots, \phi^*_{n+i_l}(|s_l|)\rangle\!\rangle_n$ and p is the one given above. This obviously holds by the definition of β_ϕ.

4.2 Commutativity

In CRSs, there are two kinds of variables, i.e. "variables" and "metavariables". Accordingly, there are two kinds of substitutions:

- substitution of variables (written as β in Lemma 7), to perform (essentially) the β-reduction of an instantiated meta-application, such as an instance of F[x].
- substitution of metavariables (written as θ in Lemma 8), used to instantiate rewrite rules, and formally called valuation (Def. 3).

The labelling map ϕ^{L} has to commute with these two substitutions. This is needed is that to establish higher-order semantic labelling. We translate a usual rewrite $s \to_{\mathcal{R}} t$ to the labelled rewrite $\phi^{\mathsf{L}}_n s \to_{\overline{\mathcal{R}}} \phi^{\mathsf{L}}_n t$ (Prop. 9). This process requires to push substitutions from inside to outside of an application of the labelling map in term structures in two levels (i.e. for variables and for metavariables). Mathematically, this is commutativity of labelling with substitutions.

Lemma 7. *Let $\phi : 0 \longrightarrow M$ be an assignment. Then, the following diagram commutes in $\boldsymbol{Set}^{\mathbb{F}}$:*

$$\begin{tikzcd}
M_\Sigma 0 \bullet M_\Sigma 0 \arrow[r, "\beta"] \arrow[d, "\phi^{\mathsf{L}} \bullet \phi^{\mathsf{L}}"'] & M_\Sigma 0 \arrow[d, "\phi^{\mathsf{L}}"] \\
M_{\overline{\Sigma}} 0 \bullet M_{\overline{\Sigma}} 0 \arrow[r, "\beta_\phi"'] & M_{\overline{\Sigma}} 0
\end{tikzcd}$$

Proof. Since ϕ^{L} is a Σ-monoid morphism, it preserves the multiplication.

Lemma 8. *Let $\phi : 0 \longrightarrow M$ and $\theta : Z \longrightarrow M_\Sigma 0$ be assignments. Then, the following diagram commutes in $\boldsymbol{Set}^{\mathbb{F}}$:*

$$\begin{tikzcd}
M_\Sigma Z \arrow[r, "\theta^*"] \arrow[d, "(\phi^*\theta)^{\mathsf{L}}"'] & M_\Sigma 0 \arrow[d, "\phi^{\mathsf{L}}"] \\
M_{\overline{\Sigma}} Z \arrow[r, "\overline{(\phi^{\mathsf{L}}\theta)}^*"'] & M_{\overline{\Sigma}} 0
\end{tikzcd}$$

Here $(-)^{\overline{*}}$ denotes the $\overline{\Sigma}$-monoid morphism extension $(-)^{*}$ (cf. Def. 3) for the case of the labelled signature $\overline{\Sigma}$.

4.3 Labelled System

For a given CRS $(\Sigma, \mathcal{R})$ and Σ-monoid M, we define the labelled rules by

$$\overline{\mathcal{R}} = \{Z \vdash \vec{n}.\phi^{\mathsf{L}}_{n} l \to \vec{n}.\phi^{\mathsf{L}}_{n} r \mid Z \vdash \vec{n}.l \to \vec{n}.r \in \mathcal{R},\ \text{assignment}\ \phi : Z \longrightarrow M\}.$$

Thus $\overline{\mathcal{R}}$ is a set of rewrite rules on labelled terms in $M_{\overline{\Sigma}}Z(0)$. So, $(\overline{\Sigma}, \overline{\mathcal{R}})$ forms a CRS that gives rewriting on $\overline{\Sigma}$-terms. We have seen that the labelling map ϕ^{L} is a Σ-monoid morphism, i.e., preserves Σ-meta-term structures. The following proposition states that ϕ^{L} moreover preserves $\mathcal{R}$-rewrite structures.

Proposition 9. *Let M be a model of $\mathcal{R}$. If we have CRS rewriting $n \vdash s \to_{\mathcal{R}} t$ on $M_{\Sigma}0_n$, then for the assignment $\phi : 0 \longrightarrow M$, we have rewriting $n \vdash \phi^{\mathsf{L}}_{n} s \to_{\overline{\mathcal{R}}} \phi^{\mathsf{L}}_{n} t$ on $M_{\overline{\Sigma}}0_n$.*

Theorem 10 (Higher-order semantic labelling). *Let M be a model of $\mathcal{R}$. A CRS $\mathcal{R}$ is terminating if and only if $\overline{\mathcal{R}}$ is terminating.*

Proof. For both directions, we prove contrapositions. [$\Leftarrow$]: By Prop. 9. [$\Rightarrow$]: By erasing all labels in rewrite steps. □

4.4 Example

We illustrate how to apply the higher-order semantic labelling method. Higher-order semantic labelling itself merely transforms a CRS into a labelled one. We need separately a way to prove termination of the labelled system. For this purpose, we use Blanqui's version of the General Schema for CRSs [Bla00] to prove termination of labelled CRSs because in our experience, this is the most powerful decidable method to prove termination of CRSs. The General Schema uses a *precedence* which is a partial order on function symbols occurring in a CRS. Using a precedence, if all rewrite rules of a given CRSs follows the General Schema, we conclude termination of it.

Example 11 (CRS for prefix sum). Consider the example of CRS $\mathcal{P}$ for computing prefix sum of lists given in Example 1. The CRS $\mathcal{P}$ is formulated under the binding signature $\Sigma = \{\mathsf{map} : \langle 1, 0\rangle, \mathsf{S}, \mathsf{ps} : \langle 0\rangle, 0, \mathsf{nil} : \langle\rangle, +, \text{“}:\text{”} : \langle 0, 0\rangle\}$.

To use higher-order semantic labelling, we need a model of $\mathcal{P}$. Here we take the presheaf $\mathcal{M}_n \triangleq (\mathbb{N}^n \to \mathbb{N})$ of all functions on $\mathbb{N}$. This $\mathcal{M}$ forms a monoid in the monoidal category $\mathbf{Set}^{\mathbb{F}}$ by taking the multiplication $\beta : \mathcal{M} \bullet \mathcal{M} \to \mathcal{M}$ as the composition “$\circ$”, and the unit $\nu : \mathrm{V} \to \mathcal{M}$ as the projections of Cartesian products $i \mapsto \pi_i$. To construct a Σ-monoid $\mathcal{M}$, we define a Σ-algebra structure on $\mathcal{M}$. First, we define the operations at the stage 0 (here we call the component parameter of a natural transformation *stage*):

$$\mathsf{map}_{\mathcal{M}_0}(f, y) = y \quad \mathsf{ps}(x) = x \quad :_{\mathcal{M}_0}(x, y) = y + 1 \quad \mathsf{nil}_{\mathcal{M}_0} = 0 \quad x +_{\mathcal{M}_0} y = 0.$$

The idea of this model is to count the number of cons's. The definition of $:_{\mathcal{M}_0}$ reflects this idea and the definition of $\mathsf{map}_{\mathcal{M}_0}$ comes from the observation that map does not change the number of cons's. For each $f : \langle i_1, \ldots, i_l \rangle \in \Sigma$, the operation at stage $n \geq 1$ is given by using pairing of functions $f_{\mathcal{M}_n}(a_1, \ldots, a_l) \triangleq f_{\mathcal{M}_0} \circ \langle a_1, \ldots, a_l \rangle$, more concretely, $f_{\mathcal{M}_n}(a_1, \ldots, a_l)(\Gamma) = f_{\mathcal{M}_0}(a_1(\Gamma), \ldots, a_l(\Gamma))$ for $\Gamma \in \mathbb{N}^n$. This indeed gives a morphism of $\mathbf{Set}^{\mathbb{F}}$. We can straightforwardly check that this gives a model of $\mathcal{P}$. We label the function symbol ps and assume that other function symbols are unlabelled. We use the natural numbers $\mathbb{N}$ as the semantic label set S_{ps}. The semantic label map is defined by $\langle\!\langle x \rangle\!\rangle_0^{\mathsf{ps}} = x$. Then, we have the following labelled rules

$$\mathsf{ps}_0(\mathsf{nil}) \to \mathsf{nil}$$
$$\mathsf{ps}_{i+1}(\mathrm{x} : \mathrm{xs}) \to \mathrm{x} : \mathsf{ps}_i(\mathsf{map}(a.\mathrm{x} + a, \mathrm{xs}))$$

for all $i \in \mathbb{N}$. the General Schema succeeds in showing termination of this labelled CRS with the precedence $\mathsf{ps}_i > \mathsf{ps}_j > \mathsf{map} > \mathsf{nil}, :$ for $i > j \in \mathbb{N}$.

5 Labelling with Quasi-models

Until now the model M was a presheaf and semantic label set S_f was a set. Here we require them to be equipped with well-founded partial orders. The operations $f_{\mathcal{M}}$ and semantic label map $\langle\!\langle - \rangle\!\rangle_f$ have to be weakly monotone morphisms in $\mathbf{Set}^{\mathbb{F}}$. Moreover, here M is only required to be a quasi-model for a CRS, meaning that the interpretation of the left-hand side of a rule is greater than or equal to ($\geq$) the corresponding right-hand side.

We define this labelling with quasi-models formally. For $f : \langle i_1, \ldots, i_l \rangle \in \Sigma$, we associate a well-founded poset $(S_f, \geq_S)$ of *semantic labels* and a *semantic label map* that is a weakly monotone morphism $\langle\!\langle - \rangle\!\rangle^f : \delta^{i_1} M \times \cdots \times \delta^{i_l} M \longrightarrow K_{S_f}$. The labelled signature $\overline{\Sigma}$ is defined by using the semantic label set S_f as in Sec. 4. Let $(M, \geq_M)$ be a quasi-model for a CRS $\mathcal{R}$. Using the semantic label map and the Σ-monoid M, the labelled CRS $\overline{\mathcal{R}}$ is also defined by the same as in Sec. 4.3. Moreover, we define the CRS Decr (called "decreasing rules") over $\overline{\Sigma}$ to consist of the rules

$$f_p(\vec{i_1}.\mathrm{z}_1[\vec{i_1}], \ldots, \vec{i_1}.\mathrm{z}_l[\vec{i_l}]) \;\to\; f_q(\vec{i_1}.\mathrm{z}_1[\vec{i_1}], \ldots, \vec{i_1}.\mathrm{z}_l[\vec{i_l}])$$

for all $f : \langle i_1, \ldots, i_l \rangle \in \Sigma$ and all $p >_S q \in S_f$. Here each metavariable z_k has arity i_k (for $1 \leq k \leq l$) and $>_S$ denotes the strict part of $\geq_S$.

Proposition 12. *Let $(M, \geq_M)$ be a quasi-model for $\mathcal{R}$. If we have rewriting $n \vdash s \to_{\mathcal{R}} t$ on $M_\Sigma 0_n$, then for the assignment $\phi : 0 \longrightarrow M$, $n \vdash \phi_n^L s \to_{\mathsf{Decr}}^*; \to_{\overline{\mathcal{R}}} \phi_n^L t$ holds. Here ";" denotes the sequential composition of relations.*

Theorem 13. *Let M be a quasi-model for a CRS $\mathcal{R}$ and $\overline{\mathcal{R}}$ the labelled CRS with respect to M. Then $\mathcal{R}$ is terminating if and only if $\overline{\mathcal{R}} \cup \mathsf{Decr}$ is terminating.*

Proof. For both directions, we prove contrapositions. [$\Leftarrow$]: By Prop. 12. [$\Rightarrow$]: By erasing all labels in rewrite steps. □

Example 14 (CRS for quick sort). Quick sort algorithm on natural numbers can be implemented as the CRS $\mathcal{R}$ with the standard rewrite rules: if, $+\!\!+$, filter, "$>$", "$\leq$".

$$\begin{array}{llll}
0 > \mathrm{Y} & \to \mathsf{false} & 0 \leq \mathrm{Y} & \to \mathsf{true} \\
\mathrm{X} > 0 & \to \mathsf{true} & \mathsf{s}(\mathrm{X}) \leq 0 & \to \mathsf{false} \\
\mathsf{s}(\mathrm{X}) > \mathsf{s}(\mathrm{Y}) & \to \mathrm{X} > \mathrm{Y} & \mathsf{s}(\mathrm{X}) \leq \mathsf{s}(\mathrm{Y}) & \to \mathrm{X} \leq \mathrm{Y} \\
\mathsf{if}(\mathsf{true}, \mathrm{X}, \mathrm{Y}) & \to \mathrm{X} & \mathsf{nil} +\!\!+ \mathrm{YS} & \to \mathrm{YS} \\
\mathsf{if}(\mathsf{false}, \mathrm{X}, \mathrm{Y}) & \to \mathrm{Y} & (\mathrm{X} : \mathrm{XS}) +\!\!+ \mathrm{YS} & \to \mathrm{X} : (\mathrm{XS} +\!\!+ \mathrm{YS}) \\
\mathsf{filter}(\mathrm{P}, \mathsf{nil}) & \to \mathsf{nil} & &
\end{array}$$

$$\begin{aligned}
\mathsf{filter}(\mathrm{P}, \mathrm{X} : \mathrm{XS}) &\to \mathsf{if}(\mathrm{P}[\mathrm{X}], \mathrm{X} : \mathsf{filter}(\mathrm{P}, \mathrm{XS}), \mathsf{filter}(\mathrm{P}, \mathrm{XS})) \\
\mathsf{qsort}(\mathsf{nil}) &\to \mathsf{nil} \\
\mathsf{qsort}(\mathrm{X} : \mathrm{XS}) &\to \mathsf{qsort}(\mathsf{filter}(a.\ a \leq \mathrm{X}, \mathrm{XS})) +\!\!+ ((\mathrm{X} : \mathsf{nil}) +\!\!+ \\
&\qquad \mathsf{qsort}(\mathsf{filter}(a.\ a > \mathrm{X}, \mathrm{XS})))
\end{aligned}$$

Since the argument of qsort in the right-hand side of the last rule ($\mathsf{filter}(\cdots)$) is structurally bigger than the argument of qsort in the left-hand side ($\mathrm{X} : \mathrm{XS}$), the General Schema is not applicable. The higher-order recursive path ordering for the corresponding rewrite system written in the format called Inductive Data Type Systems [BJO02] also fails [BR01].

Here, we use higher-order semantic labelling with a quasi-model. Let $D = \mathbb{N} \times \mathbb{N}^*$ with the order $\langle n, l\rangle \geq \langle n', l'\rangle \overset{def}{\iff} n \geq n'$. We use the carrier $\mathcal{M}_k \triangleq (D^k \to D)$. The operations at stage 0 are:

$$\mathsf{true}_{\mathcal{M}_0} = \langle 1, \epsilon\rangle \qquad \mathsf{false}_{\mathcal{M}_0} = \langle 0, \epsilon\rangle \qquad 0_{\mathcal{M}_0} = \langle 0, \epsilon\rangle \qquad \mathsf{s}_{\mathcal{M}_0}(\langle n, l\rangle) = \langle n+1, l\rangle$$

$$>_{\mathcal{M}_0} (m, n) = \begin{cases} \langle 1, \epsilon\rangle & \text{if } m > n \\ \langle 0, \epsilon\rangle & \text{otherwise} \end{cases} \quad \leq_{\mathcal{M}_0} (m, n) = \begin{cases} \langle 1, \epsilon\rangle & \text{if } m \leq n \\ \langle 0, \epsilon\rangle & \text{otherwise} \end{cases}$$

$$\mathsf{qsort}_{\mathcal{M}_0}(\langle n, l\rangle) = \langle n, \epsilon\rangle \qquad \mathsf{if}_{\mathcal{M}_0}(b, y, z) = \begin{cases} y & \text{if } b = \langle 1, \epsilon\rangle \\ z & \text{if } b = \langle 0, \epsilon\rangle \\ \langle 0, \epsilon\rangle & \text{otherwise} \end{cases}$$

$$\mathsf{filter}_{\mathcal{M}_0}(p, \langle n, l\rangle) = \langle \text{the number of } p(\langle i, \epsilon\rangle) = \langle 1, \epsilon\rangle \text{ for every } i \text{ in } l, \epsilon\rangle$$

$$+\!\!+_{\mathcal{M}_0} (\langle n, l\rangle, \langle n', l'\rangle) = \langle n + n', l \cdot l'\rangle$$

$$\mathsf{nil}_{\mathcal{M}_0} = \langle 0, \epsilon\rangle \qquad :_{\mathcal{M}_0}(\langle a, s\rangle, \langle n, l\rangle) = \langle n+1, a \cdot l\rangle$$

The operations at stage $n > 0$ are defined similarly to Example 11. This is indeed a quasi-model and cannot be a model. We label the function symbol qsort only. The semantic label set S_{qsort} is $\mathbb{N}$ with the usual order. The semantic label map is $\langle\!\langle \langle n, l\rangle \rangle\!\rangle_0^{\mathsf{qsort}} = n$, which is weakly monotone. Then, we have the labelled rules:

$$\begin{aligned}
\mathsf{qsort}_0(\mathsf{nil}) &\to \mathsf{nil} \\
\mathsf{qsort}_{i+1}(\mathrm{X} : \mathrm{XS}) &\to \mathsf{qsort}_j(\mathsf{filter}(a.a \leq \mathrm{X}, \mathrm{XS})) +\!\!+ ((\mathrm{X} : \mathsf{nil}) +\!\!+ \\
&\qquad \mathsf{qsort}_k(\mathsf{filter}(a.a > \mathrm{X}, \mathrm{XS}))) \quad \text{where } i+1 > j, k \\
\mathsf{qsort}_i(\mathrm{XS}) &\to \mathsf{qsort}_j(\mathrm{XS}) \qquad \text{for all } i > j \in \mathbb{N}
\end{aligned}$$

the General Schema shows termination of the labelled CRS with the precedence $\mathsf{qsort}_i > \mathsf{qsort}_j > \mathsf{filter} > \mathsf{if}, +\!\!+, \text{“}>\text{”}, \text{“}\leq\text{”} > \mathsf{nil}, :, 0, \mathsf{S}, \mathsf{true}, \mathsf{false}$ for $i > j \in \mathbb{N}$.

Example 15 (Haskell's rewrite rule for map/map). Consider the CRS's rewrite rule in Example 2. We take the same carrier $\mathcal{M}^n = (\mathbb{N}^n \to \mathbb{N})$ as in Example 14. Now the operation on $\mathcal{M}$ is $\mathsf{map}_{\mathcal{M}_0} : (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \longrightarrow \mathbb{N}$, $\mathsf{map}_{\mathcal{M}_0}(f, x) = x + 1$. The $\mathcal{M}$ is indeed a quasi-model. We use the semantic label set and semantic label map for map are $S_{\mathsf{map}} = \mathbb{N}$ with the usual order, and $\langle\!\langle f, x \rangle\!\rangle_0^{\mathsf{map}} = x$, which is weakly monotone. This gives the labelled rules given in Introduction, hence the original rule terminates.

6 Conclusion

We have given a method of proving termination of higher-order rewrite rules in Klop's format called combinatory reduction system (CRS). The method to prove termination, called higher-order semantic labelling, is an extension of a method known in the theory of term rewriting. This attaches semantics of the arguments to each function symbol. We systematically define the labelling by using the complete algebraic semantics of CRS, Σ-monoids. A key to establish the main theorem of semantic labelling was commutativity of labelling with two kinds of substitutions appearing in formulation of CRS. We have examined the power of higher-order semantic labelling by several examples taken from functional programming. This shows usefulness of higher-order semantic labelling in programming languages.

Acknowledgments. I am grateful to the anonymous referees for useful comments on improving the presentation of the paper. This work is supported by the JSPS Grant-in-Aid for Scientific Research (19700006).

References

[Acz78] Aczel, P.: A general Church-Rosser theorem. Technical report, University of Manchester (1978)

[BJO02] Blanqui, F., Jouannaud, J.-P., Okada, M.: Inductive data type systems. Theoretical Computer Science 272, 41–68 (2002)

[BJR08] Blanqui, F., Jouannaud, J.-P., Rubio, A.: The computability path ordering: The end of a quest. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 1–14. Springer, Heidelberg (2008)

[Bla00] Blanqui, F.: Termination and confluence of higher-order rewrite systems. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 47–61. Springer, Heidelberg (2000)

[BN98] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)

[BR01] Borralleras, C., Rubio, A.: A monotonic higher-order semantic path ordering. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 531–547. Springer, Heidelberg (2001)

[dB72] de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae 34, 381–391 (1972)

[DR98] Danvy, O., Rose, K.H.: Higher-order rewriting and partial evaluation. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 286–301. Springer, Heidelberg (1998)

[FPT99] Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: Proc. 14th Annual Symposium on Logic in Computer Science, pp. 193–202 (1999)

[Ham04] Hamana, M.: Free Σ-monoids: A higher-order syntax with metavariables. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 348–363. Springer, Heidelberg (2004)

[Ham05] Hamana, M.: Universal algebra for termination of higher-order rewriting. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 135–149. Springer, Heidelberg (2005)

[Ham07] Hamana, M.: Higher-order semantic labelling for inductive datatype systems. In: Ninth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2007), pp. 97–108 (2007)

[JR06] Jouannaud, J.-P., Rubio, A.: Higher-order orderings for normal rewriting. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 387–399. Springer, Heidelberg (2006)

[JR07] Jouannaud, J.-P., Rubio, A.: Polymorphic higher-order recursive path orderings. Journal of ACM 54(1) (2007)

[JTH01] Jones, S.P., Tolmach, A., Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in GHC. In: Haskell Workshop 2001 (2001)

[Klo80] Klop, J.W.: Combinatory Reduction Systems. PhD thesis, CWI, Amsterdam. Mathematical Centre Tracts, vol. 127 (1980)

[KOR93] Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: Introduction and survey. Theor. Comput. Sci. 121(1&2), 279–308 (1993)

[Mac71] Mac Lane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, vol. 5. Springer, New York (1971)

[Nip91] Nipkow, T.: Higher-order critical pairs. In: Proc. 6th IEEE Symp. Logic in Computer Science, pp. 342–349 (1991)

[Pol94] van de Pol, J.: Termination proofs for higher-order rewrite systems. In: Heering, J., Meinke, K., Möller, B., Nipkow, T. (eds.) HOA 1993. LNCS, vol. 816, pp. 305–325. Springer, Heidelberg (1994)

[Raa01] van Raamsdonk, F.: On termination of higher-order rewriting. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 261–275. Springer, Heidelberg (2001)

[Ter03] Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)

[Zan95] Zantema, H.: Termination of term rewriting by semantic labelling. Fundamenta Informaticae 24(1/2), 89–105 (1995)

A Appendix

A.1 Proof of Lemma 8

By induction on meta-terms in $M_\Sigma Z$. The cases x and $f(\vec{s}) \in M_\Sigma Z_n$ are straightforward. For the case $\mathrm{z}[\vec{t}] \in M_\Sigma Z_n$, we have the following.

$$\begin{aligned}
\text{lhs} &= \phi^{\llcorner}\theta^{*}(\mathrm{z}[\vec{t}]) \\
&= \phi^{\llcorner}\beta(\theta\mathrm{z};\theta^{*}\vec{t}) = \beta_{\phi}(\phi^{\llcorner}\theta\mathrm{z};\phi^{\llcorner}\theta^{*}\vec{t}) \qquad \text{(by Lemma 7)} \\
\text{rhs} &= (\phi^{\llcorner}\theta)^{\overline{*}}(\phi^{*}\theta)^{\llcorner}\mathrm{z}[\vec{t}] \\
&= (\phi^{\llcorner}\theta)^{\overline{*}}\mathrm{z}[(\phi^{*}\theta)^{\llcorner}\vec{t}] \\
&= \beta_{\phi}(\phi^{\llcorner}\theta\mathrm{z};(\phi^{\llcorner}\theta)^{\overline{*}}(\phi^{*}\theta)^{\llcorner}\vec{t}) \\
&= \beta_{\phi}(\phi^{\llcorner}\theta\mathrm{z};\phi^{\llcorner}\theta^{*}\vec{t}) = \text{lhs} \qquad \text{(by I.H.)}
\end{aligned}$$

A.2 Proof of Proposition 9

By induction on proof trees of $\to_{\mathcal{R}}$. Since $\mathcal{R}$ is structural, it suffices to consider the following two cases [Ham05].

(i) Case $n \vdash \theta_n^* l \to_{\mathcal{R}} \theta_n^* r$.
This is derived from $Z \vdash \vec{n}.l \to \vec{n}.r \in \mathcal{R}$ where $\theta : Z \longrightarrow M_{\Sigma}0$. Let $\phi : 0 \longrightarrow M$ be the assignment. Now we have a labeled rule

$$(\phi^*\theta)_n^{\mathsf{L}} l \to (\phi^*\theta)_n^{\mathsf{L}} r \in \overline{\mathcal{R}}.$$

By Lemma 8 and closedness of $\overline{\mathcal{R}}$-rewrite by the valuation $\phi^{\mathsf{L}}\theta : Z \longrightarrow M_{\overline{\Sigma}}0$, we have

$$\phi_n^{\mathsf{L}}(\theta_n^* l) = (\phi^{\mathsf{L}}\theta)_n^{\overline{*}}(\phi^*\theta)_n^{\mathsf{L}} l \ \to_{\overline{\mathcal{R}}} \ (\phi^{\mathsf{L}}\theta)_n^{\overline{*}}(\phi^*\theta)_n^{\mathsf{L}} r = \phi_n^{\mathsf{L}}(\theta_n^* r)$$

(ii) Case $n \vdash f(\ldots, n+\vec{i}.s, \ldots) \to_{\mathcal{R}} f(\ldots, n+\vec{i}.t, \ldots)$.
This is derived from $n+i \vdash s \to_{\mathcal{R}} t$. Since M is a model, notice $\phi_{n+i}^* s = \phi_{n+i}^* t$. By induction hypothesis, we have $\phi_{n+i}^{\mathsf{L}} s \to_{\overline{\mathcal{R}}} \phi_{n+i}^{\mathsf{L}} t$. So,

$$\begin{aligned}
&\phi_n^{\mathsf{L}}(f(\ldots, n+\vec{i}.s, \ldots)) \\
&= f_{\langle\!\langle \ldots, \phi_{n+i}^* s, \ldots \rangle\!\rangle_n}(\ldots, n+\vec{i}.\phi_{n+i}^{\mathsf{L}} s, \ldots) \\
&= f_{\langle\!\langle \ldots, \phi_{n+i}^* t, \ldots \rangle\!\rangle_n}(\ldots, n+\vec{i}.\phi_{n+i}^{\mathsf{L}} s, \ldots) \\
&\to_{\overline{\mathcal{R}}} f_{\langle\!\langle \ldots, \phi_{n+i}^* t, \ldots \rangle\!\rangle_n}(\ldots, n+\vec{i}.\phi_{n+i}^{\mathsf{L}} t, \ldots) \\
&= \phi_n^{\mathsf{L}}(f(\ldots, n+\vec{i}.t, \ldots))
\end{aligned}$$

A.3 Proof of Proposition 12

By induction on proof trees of $\to_{\mathcal{R}}$.

(i) Case $n \vdash \theta_n^* l \to_{\mathcal{R}} \theta_n^* r$. This case is proved by the same as in the proof of Prop. 9.
(ii) Case $n \vdash f(\ldots, n+i.s, \ldots) \to_{\mathcal{R}} f(\ldots, n+i.t, \ldots)$
This is derived from $n+i \vdash s \to_{\mathcal{R}} t$. Since $(M, \geq_M)$ is a quasi-model, we have $\phi_{n+i}^* s \geq_{M(n+i)} \phi_{n+i}^* t$. By induction hypothesis, we have $\phi_{n+i}^{\mathsf{L}} s \to_{\mathsf{Decr}}^*; \to_{\overline{\mathcal{R}}} \phi_{n+i}^{\mathsf{L}} t$. Notice also that $\langle\!\langle - \rangle\!\rangle$ is weakly monotone. So,

$$\begin{aligned}
\phi_n^{\mathsf{L}}(f(\ldots, n+i.s, \ldots)) &= f_{\langle\!\langle \ldots, \phi_{n+\vec{i}}^* s, \ldots \rangle\!\rangle_n}(\ldots, n+\vec{i}.\phi_{n+i}^{\mathsf{L}} s, \ldots) \\
&\to_{\mathsf{Decr}}^* f_{\langle\!\langle \ldots, \phi_{n+i}^* t, \ldots \rangle\!\rangle_n}(\ldots, n+\vec{i}.\phi_{n+i}^{\mathsf{L}} s, \ldots) \\
&\to_{\mathsf{Decr}}^*; \to_{\overline{\mathcal{R}}} f_{\langle\!\langle \ldots, \phi_{n+i}^* t, \ldots \rangle\!\rangle_n}(\ldots, n+\vec{i}.\phi_{n+i}^{\mathsf{L}} t, \ldots) \\
&= \phi_n^{\mathsf{L}}(f(\ldots, n+\vec{i}.t, \ldots))
\end{aligned}$$

A.4 Structural CRSs as Typed CRSs

In [Bla00], Blanqui defined a version of higher-order rewriting format Inductive Data Type Systems (IDTS), which he called "new definition of IDTS" ([Bla00] Def. 1). We call his “new definition of IDTS” *typed CRS* since as mentioned in his paper, it is a simply-typed version of CRS. Blow we show that our structural CRSs is a subclass of Blanqui’s typed CRSs. Hence we can apply General Schema for typed CRSs given in [Bla00] to structural CRSs to show termination of structural CRSs.

To give a typed CRSs, the following alphabet $\mathcal{A}$ ([Bla00] Def. 1) is required. In typed CRSs, types are simple types generated by the base types. (i) a set of base types, (ii) type-indexed collection of variables, (iii) type-indexed collection of function symbols, (iv) type-indexed collection of metavariables. Then the set of all meta-terms of a typed CRS is constructed from $\mathcal{A}$, and a typed CRS is a set of pairs of meta-terms.

Suppose that a structural CRS $(\Sigma, \mathcal{R})$ using a $\mathbb{N}$-indexed set Z of metavariables is given. We show that this gives rise to the following alphabet $\mathcal{A}$ and typed CRS. We assume the only base type ι and all variables (now, natural numbers) have the base type. For each function symbol $f : \langle i_1, \ldots, i_l \rangle \in \Sigma$, we assign to the type $f : \iota^{i_1}, \ldots, \iota^{i_l} \to \iota$ where $\iota^i = (\iota \to \cdots \to \iota) \to \iota$ (the part $(\iota \to \cdots \to \iota)$ denotes i-times ι). For each metavariable z of arity n in Z, we associate a metavariable z in $\mathcal{A}$ of the type $\iota^n \to \iota$. Then, the set of all structural meta-terms $\bigcup_{k \in \mathbb{N}} M_\Sigma Z(k)$ is equal to the set of all meta-terms of typed CRS given in [Bla00] under this alphabet $\mathcal{A}$. Thus, the structural CRS $\mathcal{R}$ is a typed CRS. Valuations and generation of a rewrite relation for structural CRSs also fit into those of typed CRS version.

A Taxonomy of Some Right-to-Left String-Matching Algorithms

Manuel Hernández

Universidad Tecnológica de la Mixteca
Huajuapan de León, Oaxaca, México
manuelhg@mixteco.utm.mx

Abstract. This paper presents a taxonomy of some exact, right-to-left, string-matching algorithms. The taxonomy is based on results obtained by using logic program transformation over a naive and nondeterministic specification. A derivation of the search part and some notes about the preprocessing part of each algorithm is presented. The derivations show several design decisions behind each algorithm, and allow us to organize the algorithms within a taxonomic tree, giving us a better understanding of these algorithms and possible mechanical procedures to derive them.

1 Introduction

Taxonomies of algorithms play an important role in computer programming [Par76, Dar78, Lau89, WZ96]: First, taxonomies of algorithms allow us to uncover the rationale behind the design of each algorithm and to know better how an algorithm works; second, these taxonomies can increase our comprehension of how an algorithm can efficiently be implemented or automatically derived; third, a taxonomy can also reveal certain faults in the design of an algorithm; finally, we can even discover minor variants as by-products of our main branches or developments. We will show how program transformation is not only a valuable tool to improve programs, but it also can be a useful tool for a better understanding of existing algorithms. This paper presents a taxonomy, based on logic program transformation, of some exact and right-to-left string-matching algorithms.

There are two main groups of string-matching algorithms classified according to a pair of schedules of character comparisons: either from left to right or from right to left. The Knuth–Morris–Pratt (KMP) algorithm [KMP77] belongs to the first group. The search part of the KMP algorithm has been the subject of intensive study in program derivation [CD89, Smi91, SGJ96]. However, this algorithm is hardly used in practice. Instead, string searching is often implemented following the Boyer–Moore (BM) algorithm [Ste94] or some variant; these algorithms belong to the second group. This paper shows how to obtain BM-like programs from naive and nondeterministic logic programs, through transformational methods and by using the powerset construction together with deterministic unfolding and constraint introduction.

Overview. We assume from our readers some basic familiarity with logic programming [Llo87] and logic program transformation [PP98]. An overview of this paper is as follows. Section 2 introduces the string-matching problem and describes the BM algorithm. Section 3 gives some logic program transformation techniques and the main points of our derivations. Section 4 is devoted to derive the BM algorithm and some of its variants. Section 5 presents related work and Section 6 gives some conclusions.

2 The Boyer–Moore Algorithm

The exact *string-matching problem.* The *exact string-matching problem* consists in finding the occurrences (if any) of a string called the *pattern* within another string called the *text*. Formally, let $\mathcal{A}$ be a non-empty and finite set called the *alphabet*. The elements of $\mathcal{A}$ are called *characters* or *symbols*. The set $\mathcal{A}^*$ consists of finite sequences of characters and these sequences are called *strings*. We denote by $\mathrm{Set}(S)$ the set of elements occurring in the string S, and by $\#(S)$ its length.

Let S_1 and S_2 be two strings in $\mathcal{A}^*$. The string S_1 is a *substring* of S_2 if S_1 is a subsequence of S_2. We define a partial order relation in $\mathcal{A}^*$ as follows: $S_1 \trianglelefteq S_2$ if S_1 is a *substring* of S_2. If $S_1 \trianglelefteq S_2$ we say that S_1 *occurs* in S_2. Let S, T be strings. The *concatenation of S and T*, $S +\!\!+ T$, is the new string ST. The set $\mathcal{A}^*$ endowed with the operation $+\!\!+$ is a non-commutative monoid with a unit element called the *empty word*, ϵ. A *border* of a string S is a non-empty string U such that $S = T_1U = UT_2$. When we refer to *the* border of a string S as that having the maximum length among borders of S. Given the strings S_1 and S_2, in the exact string-matching problem we want to know whether $S_1 \trianglelefteq S_2$. This is equivalent to know if there are two strings U and V such that $S_2 = US_1V$. This specification of occurrence is nondeterministic, because U and V are not determined. In case of $U = \epsilon$, S_1 occurs as a *prefix* in S_2. In case of $V = \epsilon$, S_1 occurs as a *suffix* in S_2. If $U = V = \epsilon$, S_1 is exactly S_2 (character-by-character).

A *configuration* between P and T is a static view of P and T placed together to make comparisons between characters. The portion of the text of size $\#(P)$ where P is placed is called a *window* of T. A *transition* is a pair of configurations. The purpose of transitions is to explain how a new configuration is reached from a previous one, given a rule to do this transition. A *matching schedule* is a sequential order (by reading a permutation from left to right) to make comparisons between characters of P and those in a window of T when trying to find P in T. The execution of a schedule is halted when a mismatch between characters occurs. The main matching schedule used in this paper is one based on comparing character-to-character from right to left within each window.

Naive string-matching algorithms. A naive left-to-right algorithm for solving the exact string-matching problem is to begin with a configuration where the pattern is placed leftmost in the text with a left-to-right matching schedule within the window. In this algorithm, if a mismatch occurs we displace the window one character to the right of the text and start the matching schedule again. If all characters of the pattern and those correspondent within the window of the text match, an occurrence has been found. This process is repeated until we do not have any new windows to explore. In the *naive right-to-left string-matching algorithm* we align the pattern and the text again from the leftmost part of the text, but we apply a right-to-left matching schedule in the current window. In both algorithms, if mismatches happen every time we almost finish comparing all characters of the pattern with those of the text, we obtain a worst-case performance, which is *quadratic* [CR94]. In practice, the expected performance of this algorithm is *linear.* We will focus our attention on right-to-left string-matching algorithms. The representative algorithm of this kind is the *Boyer–Moore* algorithm, which has a *sublinear* efficiency and is described in the next.

The Boyer–Moore algorithm. Boyer and Moore [BM77] improved the average case of the naive right-to-left string matching algorithm in a *sublinear* algorithm. Trying to match a pattern with a portion of a text, in case of a mismatch this algorithm uses two precomputed functions, based uniquely on the knowledge provided by the pattern, to carry out a shift from left to right. These functions are called the *good suffix rule* or δ_2 function and the *bad character rule* or δ_1 function. Both functions interact each other through the max function. There are several variants of the Boyer–Moore algorithm, depending on whether we use the δ_1 or the δ_2 function, or both. For example, we can use only the δ_2 function. By using enhanced versions of partial deduction, this variant was derived in [HR03]; in functional programming this variant was also derived in [Bir05]. Similarly, the variant that only uses δ_1 was derived in [MACD01] and [HR01]. However, in several variants no memory is kept of partial matchings obtained from the previous steps, as it was noticed in [AG86] and where is devised a variant of the Boyer–Moore algorithm that incorporates a *memory*, so avoiding redundant comparisons. In every variant, the search is fast on average, because in many cases the shifts are close to the length of the pattern.

We now give a description of the BM algorithm [BM77, Ste94]. We denote by R_k the k^{th} character of a string R, trying to keep k in an appropriate range (otherwise, R_k is undefined). In addition, T denotes a text of length n and P denotes the pattern of length m. We try to apply the index i to the text T and the index j to the pattern P. Consider first a mismatch between P_m and T_i. If T_i does not occur in P at all, then the pattern is shifted m characters to the right. The next comparison is then between P_m and T_{i+m}. If, on the other hand, T_i does occur in P, with rightmost occurrence in P_k, then T_i and P_k are lined up (i.e. the pattern is shifted $m-k$ characters) and the test is resumed by comparing P_m with T_{i+m-k}.

To align a particular character of the pattern with that of the text, we use the following δ_1 function:

$$\delta_1(x)=\begin{cases} m & \text{if } x \notin \text{Set}(P) \ , \\ m-k & k=\max\{j \in \mathbb{N} \mid P_j = x\} \end{cases}$$

If such a character exists in the text but does not exist in the pattern, we shift the pattern over the text the total length of the pattern, m.

Consider a match between P_m and T_i. Comparisons of pattern and text characters continue from right to left until a complete match is obtained or a mismatch at P_j and $T_{i'}$, say, occurs. In this case, the *suffix* of the pattern given by $P_{j+1}, \ldots, P_m$ is equal to the text substring $T_{i'+1}, \ldots, T_{i'+m-j}$, and $P_j \neq T_{i'}$.

If $T_{i'}$ does not occur in p at all, we can then shift the pattern $m - j$ positions to the right (just past $T_{i'}$), and the next comparison will be between P_m and $T_{i'+m}$. The text index will then be incremented by m positions.

If, on the other hand, $T_{i'}$ does occur in P, consider the rightmost such an occurrence, P_k. There are two possibilities: P_k is placed either to the left or to the right of P_j. In case P_k is to the left, then P_k and $T_{i'}$ and are lined up, and the next comparison will be between P_m and $T_{i'+m-k}$. Hence, we can use δ_1 to compute the shift of the text index in both cases of a partial match we have covered in our explanation so far. If, however, P_k is to the right of P_j, then δ_1 would yield a negative value, meaning a backwards displacement of the pattern. In this case we ignore δ_1 and shift the pattern one character to the right,which always ensure us correctness.

In case of partial matchings involving at least one character, we may shift the pattern more characters than those prescribed by δ_1. Instead of determining the occurrence within the pattern of the text character $T_{i'}$ that caused the mismatch, we can determine a *reoccurrence* of the pattern suffix already matched. In this case, $P_{j+1}, \ldots, P_m = T_{i-m+j+1}, \ldots, T_i$ and $P_j \neq T_{i-m+j}$. If the suffix $P_{j+1}, \ldots, P_m$ also appears in P as a substring $P_{j+1-k}, \ldots, P_{m-k}$, with $P_{j-k} \neq P_j$, and it is the rightmost such an occurrence, then the pattern may surely be shifted k characters to the right.

```
initializeBM($P, \delta_1, \delta_2$);    {preprocessing stage to tabulate the $\delta_1$,$\delta_2$ functions}
$i := m$; $j := m$;
while $(j > 0)$ and $(i \leq n)$ do
    if $T_i = P_j$ then  {rightmost characters coincide}
        begin    {trying to extend the matching from right to left}
            $i := i - 1$; $j := j - 1$
        end
    else    {a failed comparison between characters}
        begin
            $i := i + \max(\delta_1(t_i), \delta_2(j))$;   {shift based on precomputed functions}
            $j := m$
        end
if $j < 1$ then $i := i + 1$  {Pattern found}
        else $i := 0$    {Pattern not found}
```

Fig. 1. The Boyer–Moore algorithm

It may also happen that such a reoccurrence may "fall off" the left end of P, in case a border occurs: a suffix of $P_{j+1-k}, \ldots, P_{m-k}$ appears as a prefix of P. In this case, $k \geq j$. The definition of the δ_2 function is:

$$\begin{aligned}\delta_2(j) = \min\{k + m - j \mid k \geq 1 \text{ and } (k \geq j \text{ or } P_{j-k} \neq P_j) \\ \text{and } ((k \geq d \text{ or } P_{d-k} = P_d) \text{ for } j < d \leq m)\}\end{aligned}$$

The value of δ_2 always yields a positive shift. Hence, by obtaining the maximum of both δs, we not only avoid the possibility of a negative shift prescribed by δ_1, but also move the pattern as many characters as possible, given δ_1, δ_2 and the current information. Figure 1 shows the BM algorithm.

3 Logic Programming and Exact String Matching

Having described our main algorithm to deal with, in this section we describe some logic program transformation tools to be used in our derivations; also, in this section we give the main guidelines of these derivations.

3.1 Logic Program Transformation Tools

Equality theory. To use logic programming, we suppose the SLDNF-resolution rule, and the conventional unification algorithm. Because the unification algorithm of logic programming is greedy and hides some, perhaps useful, information, we try to replace it by some explicit *equation introduction*, which is basically an introduction of constraints. This equation introduction is logically justified by using the *standard equality theory* of first-order logic; to deal with *inequations* we suppose Clark's equality theory [Cla78]. Both of these tools are applied over Herbrand universes. We will use some restricted forms of equations: An equation $X = Y$ has *normal form* if X is a variable and Y is a term without any occurrence of X in Y (occur-check rule). An *inequation* is the negation of a normalized equation. Because equation introduction helps us to deal with several instances of variables, this tool is strongly related to the technique of partial evaluation named *bounded static variation* (also colloquially know as "The Trick"), a binding-time source-program transformation [DRK06].

Deterministic unfolding. In the unfold/fold method [PP98], when we unfold a clause C with respect to an atom q and q is defined by several clauses, we obtain several clauses again. If we simplify these resultant clauses and get only one clause, we say that the unfolding is *deterministic*. If the surviving clause is unfolded again with respect the same atom A, and we get only one clause again, and so on, we have a succession of deterministic unfolding steps. If after $n+1$ of these unfolding steps we obtain two or more clauses, without any possibility of eliminating some of them, we stop unfolding at step n, and we have a *succession of deterministic unfolding steps of size* n; this succession is a valuable tool for assuring termination when we apply the unfolding rule; moreover, this succession also avoids dealing with an over-specialized program. The unfolding rule and the "extended" folding rule (where several clauses are used for folding) are the basic components of the *disjunctive partial deduction* [PPR97]. An opportunity of simplifying clauses after unfolding is when we have in the body of the resultants clauses an *unsatisfiable set of atoms*. For example, if we have:

$$p(X) \leftarrow X = b \wedge \underline{q(X)} \tag{1}$$

$$q(X) \leftarrow X = a \tag{2}$$

$$q(X) \leftarrow X = b \tag{3}$$

by unfolding $q(X)$ in (1) (underlined subgoal) we have the two clauses: $p(X) \leftarrow X = b \wedge X = a$ and $p(X) \leftarrow X = b \wedge X = b$. The body of the first clause has the set of equations: $\{X = b, X = a\}$, which originates the following false fact: $b = a$ (when we apply the substitution $\sigma = \{X/b\}$). Hence, this clause is eliminated because it is useless in the computational process of resolution (*clause removal rule* [FPP02]). The body of the other clause is $\{X = b, X = b\} = \{X = b\}$, and then the only clause we obtain is: $p(X) \leftarrow X = b$, so that the unfolding of clause (1) with respect to q is deterministic. When we eliminate the occurrence of a subgoal q within the body of a clause by a succession of deterministic unfolding steps we say that we have applied *total unfolding* to q.

Clause splitting rule. The clause splitting rule allows us to treat complementary cases, perhaps in an exhaustive way. Given the following clause $C: \quad p \leftarrow q$ by applying the clause splitting rule [FPP02] we generate a pair of clauses: $p \leftarrow r \wedge q$ and $p \leftarrow s \wedge q$, where $r \vee s$ is equivalent to *true*. The subgoal r can be an equation, and then s is the negation of this equation (i.e., an inequation). The equations can be introduced by the equation introduction rule.

We can also apply the clause splitting rule to sets:

$$p(X) \leftarrow X \in S \wedge g(X) \tag{4}$$

$$p(X) \leftarrow X \notin S \wedge g(X) \tag{5}$$

where S is a set. If S is a finite set, to say, $S = \{a, b\}$, the clause splitting rule over S is *extended* and is expressed as:

$$p(X) \leftarrow X = a \wedge g(X) \tag{6}$$

$$p(X) \leftarrow X = b \wedge g(X) \tag{7}$$

$$p(X) \leftarrow X \neq a \wedge X \neq b \wedge g(X) \tag{8}$$

which essentially means a *total unfolding* of *member/2* and *nonmember/2*:

$$\mathit{member}(E, [A|Ls]) \leftarrow E = A \tag{9}$$

$$\mathit{member}(E, [A|Ls]) \leftarrow \mathit{member(E,Ls)} \tag{10}$$

$$\mathit{nonmember}(E, Ls]) \leftarrow \backslash\!+ \ \mathit{member(E,Ls)} \tag{11}$$

It is also possible to introduce inequations expressed as inequalities [FPP02].

Extended folding. In [GK94] we found a basic idea: we can use definitions consisting of several clauses to fold. Applied to partial deduction, this idea generated the *disjunctive* partial deduction [PPR97]. Consider the following relation: $R = \{(a,a),\ (a,b),\ (b,a)\}$ defined on $A \times A$, with A=$\{a,b\}$ This relation is nondeterministic in the first argument, because (a,X) leaves us X with several possibilities: either $X = a$, or $X = b$. A transformation over this relation allows us to speak of a *function* instead of a relation: $\Lambda R = \{(a,\{a,b\}),(b,\{a\})\}$ [BdM97]. The cost is the following: $R \subset A \times A$, but $R \subset A \times \mathcal{P}(A)$, where $\mathcal{P}(A)$ is the power set of A. This particular fact has extensively been exploited in functional programming to simulate nondeterminism having initially relations and finally functions (which are implemented in Haskell, for example) [BdM97].

In logic programming an application of the same technique results more natural because we can model relations as predicates, and we would have a logic program as the following: $\{r(a,a)\leftarrow,\ r(a,b)\leftarrow,\ r(b,a)\leftarrow\}$. By introducing equations we have:

$$r(X,Y) \leftarrow \boxed{X = a} \wedge Y = a \tag{12}$$
$$r(X,Y) \leftarrow \boxed{X = a} \wedge Y = b \tag{13}$$
$$r(X,Y) \leftarrow X = b \wedge Y = a \tag{14}$$

where clauses (12) and (13) share the subgoal $X = a$. We name this common subgoal a *pivot* and show it boxed-in. For folding, we create a new definition: $g(Y) \leftarrow Y = a$ and $g(Y) \leftarrow Y = b$. By using the pivot $X = a$ and folding wrt this new definition, we have the program consisting of clauses $r(a,Y) \leftarrow g(Y)$ and $r(b,Y) \leftarrow h(Y)$, where $h(Y) \leftarrow Y = a$ (by unfolding $g/1$ and $h/1$ in the body of these clauses we would obtain the original definition of $r/2$, which is described as *in-situ* folding.) This new definition of $r/2$ is deterministic *with respect to its first argument.* So that by using extended folding we decrease or eliminate nondeterminism to obtain deterministic implementations. This method is rooted in the powerset construction (named in [BdM97] as *Eilenberg–Wright Lemma* (p. 122)) to transform nondeterministic automata into deterministic ones.

3.2 Guidelines of Our Derivations

As we have seen, the pattern P *occurs* in a string T if there exist strings X_1 and X_2 such that $T = (X_1 +\!\!+ P) +\!\!+ X_2$. We use this specification as a guide in the following naive logic program that solves the string-matching problem:

$$\mathit{substring}(P,T) \leftarrow \mathit{append}(X_1,P,X_2) \wedge \mathit{append}(X_2,X_3,T) \tag{15}$$

where *append* has the usual definition. We should notice the existential variables X_1 and X_2, and that we associate *append*/3 to the left. Adapting our notation to logic programming, now we consider a particular pattern $P = p_1 \ldots p_n$; this pattern is represented by the list $[p_1,\ldots,p_n]$, or as $[p_1 : p_n]$, where with the notation $O_i : O_j$ we abbreviate the sequence of objects $O_i, O_{i+1},\ldots,O_j$ if $i < j$, or the sequence $O_i, O_{i-1},\ldots,O_j$ if $j < i$. If $i = j$, $O_i : O_j = O_i$. Moreover, we use $A_i : A_j = B_i : B_j$ as an abbreviation of a *conjunction* of equations: $A_i = B_i \wedge A_{i+1} = B_{i+1} \wedge \ldots \wedge A_j = B_j$ and $A_i : A_j \approx B_i : B_j$ as an abbreviation of a *conjunction* of equations: $A_i : A_{j-1} = B_i : B_{j-1}$ but $A_j \neq B_j$. Indexes i and j follow the previous convention if $i < j$ or $j > i$. Again, if $i = j$, $A_i : A_j = B_i : B_j$ is reduced to $A_i = B_i$.

From the clause (15), by using some standard techniques of logic program transformation and partial deduction, we obtain the following clauses:

$$ss_P([p_1 : p_m \,|\, L]) \leftarrow \tag{16a}$$
$$ss_P([A \,|\, L]) \leftarrow ss_P(L) \tag{16b}$$

$\rightsquigarrow$ By equation introduction

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_1 : A_m = p_1 : p_m \tag{16c}$$
$$ss_P([A \,|\, L]) \leftarrow ss_P(L) \tag{16d}$$

$\rightsquigarrow$ By reordering the equations and the most specific program

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{16e}$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow ss_P([A_2 : A_m \,|\, L]) \tag{16f}$$

This last program (16e,16f) disallows to use texts of length lesser than m. With this matching schedule, each particular mismatched character involves a suffix of the pattern (including the ϵ string). Thus, we have that either $A_m = p_m$ or $A_m \neq p_m$; and, if $A_m = p_m$ then either $A_{m-1} = p_{m-1}$ or $A_{m-1} \neq p_{m-1}$, and so on. Finally, if $A_m : A_2 = p_m : p_2$ then either $A_1 = p_1$ or $A_1 \neq p_1$. We call the program incorporating in the body of its clauses these equations and inequations *a program in triangular form.*

Program 1

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{17a}$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge ss_P([A_2 : A_m \,|\, L]) \tag{17b}$$
$$\bigwedge_{k=1}^{m-1} ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_k \approx p_m : p_k \wedge ss_P([A_2 : A_m \,|\, L]) \tag{17c}$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_1 = p_m : p_1 \wedge ss_P([A_2 : A_m \,|\, L]) \tag{17d}$$

(We use the $\bigwedge$ symbol to concisely denote the *conjuntion of clauses*, by following the declarative reading of a logic program.) Clauses (17a) and (17d) deal with a total match and possible reoccurrences of the pattern, respectively. Clause (17b) treats an initial mismatching. The other clauses find partial matchings of size $k-1$, and the constraint $A_k \neq p_k$:

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_k \approx p_m : p_k \wedge ss_P([A_2 : A_m \,|\, L]) \tag{18}$$

where k is such that $1 \leq k \leq m-1$.

In Prog. 1 the nondeterminism has been increased, but we can make the following process to decrease it. We select $A_m = p_m$ as pivot, because this subgoal is common to the body of several clauses, and define a new predicate to fold some clauses. Similarly, we select $A_{m-1} = p_{m-1}$ as the next pivot. This process gives us every pivot, consecutively. *Without any application of some other rule, apart from the folding rule*, we would get the following *cascade-like program*:

Program 2

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m = p_m \wedge new_1([A_1 : A_m \,|\, L]) \tag{19}$$

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge ss_P([A_2 : A_m \,|\, L]) \tag{20}$$

$$\bigwedge_{k=1}^{m-1} \begin{bmatrix} new_k([A_1 : A_m \,|\, L]) \leftarrow A_{m-k} = p_{m-k} \wedge new_2([A_1 : A_m \,|\, L]) \\ new_k([A_1 : A_m \,|\, L]) \leftarrow A_{m-k} \neq p_{m-k} \wedge ss_P([A_2 : A_m \,|\, L]) \end{bmatrix} \tag{21}$$

$$new_m([A_1 : A_m \,|\, L]) \leftarrow \tag{22}$$

$$new_m([A_1 : A_m \,|\, L]) \leftarrow ss_P([A_2 : A_m \,|\, L]) \tag{23}$$

Further constraints will affect the number of new predicates, but the (definition-folding) process will be the same for each variant. The main idea is to decrease the length of the size of the list $[A_2 : A_m \,|\, L]$ at each recursive call of $ss_P/1$, because when we decrease this size we increase the shift and, therefore, we decrease the number of comparisons. We will apply *deterministic unfolding* to carry out this objective, as we will see in the next section.

4 Deriving the Search Part of Some Variants of the Boyer–Moore Algorithm

Now we derive the BM algorithm and some of its variants. First, we derive variants restricted to use either the δ_1 or the δ_2 function. Next, we derive the search phase of the BM algorithm by using the maximum of both functions. We continue with the derivation of the BMH and the BMPS algorithms. Finally, we derive the BMAG algorithm. This algorithm differs from the previous ones because it incorporates a memory, so avoiding at all to access twice or more times a character text.

We define an auxiliary predicate:

$$shift(1, [A \,|\, L_1], L_1) \leftarrow \tag{24a}$$

$$shift(N, [A \,|\, L_1], L_2) \leftarrow N_1 \ is\ N - 1 \wedge shift(N_1, L_1, L_2) \tag{24b}$$

where $shift(N, L_1, L_2)$ holds when L_1 is L_2 without its first N elements. This predicate will be useful in the following.

A variant involving δ_1. We begin by applying the clause splitting rule to (16f):

Program 3

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{25a}$$

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m = p_m \wedge ss_P([A_2 : A_m \,|\, L]) \tag{25b}$$

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge ss_P([A_2 : A_m \,|\, L]) \tag{25c}$$

We notice that the negative information of $A_m \neq p_m$ in clause (25c) can be reinforced with the introduction of the constraints over sets given by $\in$ and $\notin$. Let $\mathrm{Set}(P)$ be the set of characters of the pattern $P = p_1 : p_m$, and ρ be its cardinality ($\rho \leq m$, where m is the length of the pattern). We apply the clause splitting rule to (25c), using $A_m \in \mathcal{S}_p \vee A_m \notin \mathrm{Set}(P)$, to obtain the following other two clauses:

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge A_m \in \mathrm{Set}(P) \wedge ss_P([A_2 : A_m \,|\, L]) \quad (26a)$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge A_m \notin \mathrm{Set}(P) \wedge ss_P([A_2 : A_m \,|\, L]) \quad (26b)$$

When we eliminate the subgoal ($A_m \in \mathrm{Set}(P)$) by unfolding, we get ρ new clauses; a clause, in particular, contains the following equation and inequation: $A_m \neq p_m, A_m = p_m$ in its body, and this clause is eliminated. (Note that unfolding $A_m \notin \mathrm{Set}(P)$ means to introduce ρ inequations.) In detail, we have that from the clause (26a) we get the following new clauses:

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge A_m = p_1 \wedge ss_P([A_2 : A_m \,|\, L]) \quad (27)$$
$$\vdots$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge A_{m-1} = p_{m-1} \wedge ss_P([A_2 : A_m \,|\, L]) \quad (28)$$

whereas from the clause (26b) we get the clause:

$$\begin{aligned} ss_P([A_1 : A_m \,|\, L]) \leftarrow\ & A_m \neq p_m \wedge A_m \neq p_1 \wedge \\ & \ldots \wedge A_m \neq p_m \wedge ss_P([A_2 : A_m \,|\, L]) \end{aligned} \quad (29)$$

where we can apply subgoal simplification ($A_m \neq p_m$, $A_m = p_k$, $p_k \neq p_m$ implies $A_m = p_k$).

After deterministic unfolding, for each value p_i (except for p_m, in clause (25b)), we have a correspondent shift value. These values are asserted as facts, and these facts tabulate the δ_1 function:

$$\mathit{d1}(p_1, V_1) \leftarrow \quad \ldots \quad \mathit{d1}(p_{m-1}, V_{m-1}) \leftarrow \quad \mathit{d1}(p_m, 1) \leftarrow \quad \mathit{d1}(x, m) \leftarrow$$

where x is a meta-character indicating that $x \notin \mathrm{Set}(P)$. Thus, we have the following program:

Program 4

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_1 = p_m : p_1 \quad (30a)$$
$$\begin{aligned} ss_P([A_1 : A_m \,|\, L]) \leftarrow\ & \mathit{d1}(A_m, Val) \wedge \\ & \mathit{shift}(Val, [A_2 : A_m \,|\, L], L_1) \wedge ss_P(L_1) \end{aligned} \quad (30b)$$

Because $\delta_1(p_m) = 0$ we have to deal with this case in a special form (to force a shift different from zero, Boyer and Moore put $\delta_1(p_m) = 1$). A conservative shift of one character is enough for this case: we refrain from unfolding clause (25b), according to Boyer and Moore, but deterministic unfolding could be applied without any problem to (25b).

A variant involving only δ_2*.* From the program in triangular form we can obtain a variant related to the δ_2 function. To decrease the length of $[A_2 : A_n \,|\, L]$ in the recursive call of $ss_P/1$, we use *deterministic unfolding.* Depending on whether $p_m : p_k = p_{m+1} : p_{k+1}$ and $p_{k-1} \neq p_k$ hold either we do not unfold or begin to unfold with respect to the definition of $ss_P/1$ given by clauses 16e and 16f.

We can get from the BM algorithm and its δ_2 function an analogous of the *next* table of the KMP algorithm. Let us consider the following clause:

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \wedge ss_P([A_2 : A_m \,|\, L]) \quad (31)$$

Name $A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k$ a *semi-suffix* of size k, denoted by *ssuf*(k). For each *ssuf*(k) we obtain a value Vs_k:

$$d2(ssuf(1), Vs_1) \leftarrow \qquad \ldots \qquad d2(ssuf(m-1), Vs_{m-1}) \leftarrow \qquad d2(ssuf(m), Vs_m) \leftarrow$$

Unfolding each clause of Prog. 1 we would get the δ_2 function [HR01]. By using the *shift*/3 function we obtain the following clause:

$$\begin{aligned} ss_P([A_1 : A_m \,|\, L]) \leftarrow\ & d2(ssuf(k), Vs_k) \wedge \\ & shift(Vs_k, [A_2 : A_m \,|\, L], L_1) \wedge ss_P(L_1) \end{aligned} \quad (32)$$

When we execute Prog. 2, this program would incorporate in its search phase the δ_2 function. Nondeterminism, however, has been increased. The technique to derive a deterministic program is given by the cascade-like program of Subsection 3.2, where nondeterminism is reduced, except for clause (17d), which finds reoccurrences when a pattern overlaps with itself.

The $\max(\delta_1, \delta_2)$ *function and the Boyer–Moore algorithm.* Now we analyze the BM algorithm itself. In the search phase, the BM algorithm uses $\max(\delta_1, \delta_2)$. Consider the following subgoal: $A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k$. This subgoal incorporates information about a semi-suffix of size k, and information about $A_k \neq p_k$. With respect to the semi-suffix of size k we have found a value associated with *ssuf*(k), trough the table *d2*. With respect to the inequation $A_k \neq p_k$ we also have some information, stored in table *d1*. Each shift is correct, but our objective is to have the major shift possible to get the following clause:

$$\begin{aligned} ss_P([A_1 : A_m \,|\, L]) \leftarrow\ & d1(c, Val_1) \wedge d2(ssuf(k), Val_2) \wedge \\ & \max(Val_1 - (m-k), Val_2, Val) \wedge \\ & shift(Val, [A_1 : A_m \,|\, L], L_1) \wedge ss_P(L_1) \end{aligned} \quad (33)$$

($Val_1 - (m-k)$ could be negative or zero, but when taking the maximum, the 0 value is discarded, because V_2 is always positive.)

The justification is as follows. From: $\{p \leftarrow r_1,\ p \leftarrow r_2,\ r_1 \leftarrow,\ r_2 \leftarrow\}$ we can derive a new program, $\{p \leftarrow or(r_1, r_2),\ or(r_1, r_2) \leftarrow r_1,\ or(r_1, r_2) \leftarrow r_2,\ r_1 \leftarrow,\ r_2 \leftarrow\}$ where $or(r_1, r_2)$ is a subgoal having the possibility of producing answers.

The Horspool variant. In [Hor80] it was noticed the only purpose of δ_2 is to optimize the handling of repetitive patterns and avoid the worst case. In [Hor80] a simplified and practical variang of the BM algorithm was also presented. This variant deals only with the δ_1 function, and a particular value of the δ_2 function: $\delta_2(x)|_{x=m}$, where $\delta_2(x)|_{x=m}$ means to find the rightmost occurrence of p_m. The text character that aligns with p_m is always chosen (regardless of the position where the mismatch occurred). From the clause

$$ss_P([A_1 : A_n \,|\, L]) \leftarrow A_m = p_m \wedge ss_P([A_2 : A_n \,|\, L]) \tag{34}$$

by deterministic unfolding we get $\delta_2(m)$. If we have $\delta_{12}(p_m) = \delta_2(m)$ and $\delta_{12}(p_k) = \delta_1(p_k)$, for $p_k \neq p_m$, we obtain the Horspool variant. If we do not unfold clauses related to the other arguments in the δ_2 function (BMH algorithm saves preprocessing in this part) we get the BMH algorithm.

The PS variant. After unfolding clause (26a), we get some clauses of the form:

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge A_m = p_k \wedge ss_P([A_2 : A_m \,|\, L])$$

where A_m is the rightmost character within the pattern. But the clause splitting rule can be applied to other rules having an inequation:

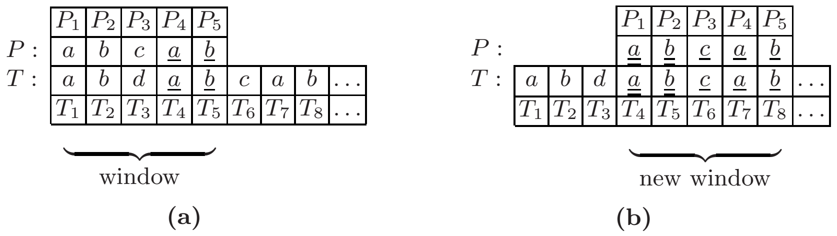
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \wedge ss_P([A_2 : A_m \,|\, L])$$

and then we have a clause of the following form:

$$\begin{aligned} ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m &: A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \\ &\wedge A_k = p_s \wedge ss_P([A_2 : A_m \,|\, L]) \end{aligned}$$

where $p_s \in \text{Set}(P)$. If we align with respect to p_s we have either some or none deterministic unfolding steps. In every case, we have to shift at least one character. On the other hand, if we do some deterministic unfolding steps in $ss_P/1$ and try to satisfy the conjunction of equations $A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k$ we get an advance related with δ_2. In fact in $xp_{k+1} : p_m$, according to Boyer and Moore, we move the pattern over the text a shift given by δ_2 or by the value associated with x (x is a meta-symbol, representing a variable). And, due to $p_s p_{k+1} : p_m$ implies $xp_{k+1} : p_m$, shifts based on $p_s p_{k+1} : p_m$ are larger than those based on the max of δ_1 and δ_2. In a certain way, this variant is more natural than the BM algorithm because we want *exact reoccurrences* of substrings. However, the preprocessing of this variant is very high (we need to consider at least $\rho * m$ distinct clauses). This variant was given in [PS90].

The BMAG variant. As pointed out in [AG86], when the BM algorithm shifts the pattern to the right, it does not retain any information about characters already matched. Thus, each previous variant (and the BM algorithm itself) makes some unnecessary comparisons. In the following variant of the BM algorithm, we keep track of substrings already matched during previous alignments, and exploit such recordings later in the matching process. With this method, no character of the text needs to be accessed more than twice. Moreover, we will see how clause (17d) helps to resume efficiently the pattern matching process following the detection of an occurrence of the pattern. At the moment of processing the recursive call of this clause we obtain the procedure devised by Galil and presented in [AG86] for detecting consecutive overlapping occurrences at once. Let us suppose the configuration given in (a), Fig. 2, where a mismatch occurs



P_1 P_2 P_3 P_4 P_5
P : a b c a b
T : a b d a b c a b ...
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 ...
window
(a)
P_1 P_2 P_3 P_4 P_5
P : a b c a b
T : a b d a b c a b ...
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 ...
new window
(b)

Fig. 2. Configurations and a transition in the BMAG algorithm

between c and d ($T_3 \neq P_3$), but the substring $ab(= P_4P_5)$ of the pattern matches with the substring $ab(= T_4T_5)$ of the text. The BM algorithm shifts the pattern from left to right, and gives us the configuration shown in (b), Fig. 2, where a total match occurs (underlined characters in the same column indicates a comparison already made). However, to find this match the BM algorithm has to make five comparisons: $T_8 : T_4 = P_5 : P_1$, whereas the BMAG algorithm only makes three comparisons: $T_8 = P_5$, $T_7 = P_4$ and $T_6 = P_3$. This is because the BMAG algorithm records the previous matching between the substrings $P_4P_5(= ab)$ and $T_4T_5(= ab)$, and does not need to make some comparisons again.

Following our approach to get a δ_2 value, we would have the following process. From clause

$$ss_P([A_1 : A_5 \,|\, L]) \leftarrow A_5 = b \wedge A_4 = a \wedge A_3 \neq c \wedge ss_P([A_2 : A_5 \,|\, L])$$

we get, by deterministic unfolding, the following one:

$$ss_P([A_1 : A_5 \,|\, L]) \leftarrow A_5 = b \wedge A_4 = a \wedge A_3 \neq c \wedge ss_P([A_4 : A_5 \,|\, L])$$

I.e., at the recursive call, character inspection begins with A_4 instead of A_2. At the recursive call, we have that

$$ss_P([A_1^{\surd}, A_2^{\surd}, A_3, A_4, A_5 \,|\, L]) \leftarrow A_5 = b \wedge A_4 = a \wedge A_3 = c \wedge A_2^{\surd} = a \wedge A_1^{\surd} = b$$

where the $\surd$ in $A_1^{\surd}$, and $A_2^{\surd}$ indicates that A_1 and A_2 are already known.

To avoid unnecessary comparisons, we treat separately each case of overlapping. In our example we continue as follows. We define a predicate ss_P^{bac}:

$$ss_P^{bac}([A_1, A_2, A_3, A_4, A_5]) \leftarrow A_5 = b \wedge A_4 = a \wedge A_3 = c$$

for avoiding to compare again the substring ab:

$$ss_P([A_1 : A_5 \,|\, L]) \leftarrow A_5 = b \wedge A_4 = a \wedge A_3 \neq c \wedge ss_P^{bac}([A_4 : A_5 \,|\, L])$$

Variables A_1 and A_2 are only used as places to be omitted in a re-scanning. Further optimizations can be achieved by applying our previous technology to ss_P^{bac} (clause splitting, deterministic unfolding, and folding, but now only to $A_5 = b \wedge A_4 = a \wedge A_3 = c$).

Because there exist at most m suffixes of a pattern of length m, we deal with at most m special cases. Even more, some suffixes (implicit in the recursive call) of the same length can be analyzed as a particular case.

Formally, to derive the BMAG algorithm, we need some means to keep track of which segments of the text matched some suffix of the pattern. In our derivation, we detect such suffixes in the recursive call of each clause after applying deterministic unfolding to such a clause:

$$ss_P([A_1 : A_m|L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \wedge \quad (35)$$
$$\mathit{shift}(d(k), [A_2 : A_m|L], L_1) \wedge ss_P(L_1) \quad (36)$$

where $d(k)$ is a displacement value ($d(k)$ is always at least 1). With a displacement of $d(k)$ we detect $d(k) - 1$ coinciding characters. The complementary part in the pattern has $m - d(k)$ characters.

$$ss_P^{p_m : p_{m-d(k)}} p([A_1 : A_m|L]) \leftarrow A_m : A_{m-d(k)} = p_m : p_{m-d(k)} \quad (37)$$

Hence, $d(k)$ characters are not more revisited. The new formulation of (36) is:

$$ss_P([A_1 : A_m|L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \wedge \quad (38)$$
$$\mathit{shift}(d(k), [A_2 : A_m|L], L_1) \wedge ss_P^{p_m : p_{m-d(k)}}(L_1) \quad (39)$$

This is called *prefix memorization* in [CR02, p.30]. Applying the clause splitting rule to the new defined predicates, we create new (sub)-triangular forms, to deal with every case of mismatching. If there exists a mismatching we call $ss_P/1$ (upper level) to deal with a possible total occurrence of the pattern. Furthermore, it is possible to do some extra deterministic unfolding steps and, finally, we can apply a systematic folding to the sub-triangular form to reduce nondeterminism.

Let us detail the procedure. First, we create new definitions:

$$\mathit{new}([B_1 : B_m \,|\, L]) \leftarrow B_m : B_{m-k} = p_m : p_{m-k} \quad (40)$$
$$\mathit{new}([B_1 : B_m \,|\, L]) \leftarrow ss_P([B_1 : B_m|L]) \quad (41)$$

We only need to analyze cases from B_{m-k} to B_m at the next call of $ss_P/1$. The next task is to define new predicates to separately treat each clause. Now we apply the clause splitting rule, but only to $B_{m-k} : B_m$; next, we apply deterministic unfolding; finally, we fold for eliminating nondeterminism.

5 Related Work

In [CD78, Dar78, RS83] the authors obtain families of several kinds of algorithms by using formal languages, but the mechanization of their derivations is left as an open problem. In our case, when dealing with string-matching algorithms, some parts of our derivations are mechanizable (mainly those parts supported by partial deduction), but some human assistance is necessary to obtain specific algorithms. In [PS90] and [Pep91] there are derivations of the search part of the BMPS variant. In [MACD01] there is a derivation of a simplified version (which includes the shifts given by the δ_1 function) of the search part of the Boyer–Moore algorithm using a naive specification equipped with a database to record comparisons. The complete search phase of the Boyer–Moore algorithm was derived in [DRK06]. In [Bir05] there is a derivation of another variant of the Boyer–Moore algorithm, relying on the definition of δ_2. In contrast to these works, we have begun with nondeterministic programs, and instead of explicit backtracking, we have taken benefit from nondeterminism of logic programming.

6 Conclusions

This paper has shown several relationships among some right-to-left string-matching algorithms (see Fig. 3) via certain design decisions and steps of logic program transformation. The final programs have some inefficiency related with the access in linear time of lists. However, it can be asserted that all our machinery performs well over specifications based on indexing; this has been shown in [HR03], at least for the BM variant restricted to use only the δ_2 function.

As future work, a proposal is to add other exact string-matching algorithms to the taxonomy presented here. In fact, at least two left-to-right algorithms can also be obtained from our techniques by altering the matching schedule: the Morris–Pratt and the Knuth–Morris–Pratt algorithms. In a similar vein, the Simon algorithm, as described in [CR94], is another candidate to be derived and added to the taxonomy. Depending on some more liberal matching schedules, some other algorithms could be included, for example, those described in [Sun90]. Other possible taxonomies related to text processing would follow the method of taking the text as static, instead of the pattern. McCreight and Ukkonen algorithms take this method and they would be good candidates to be derived.

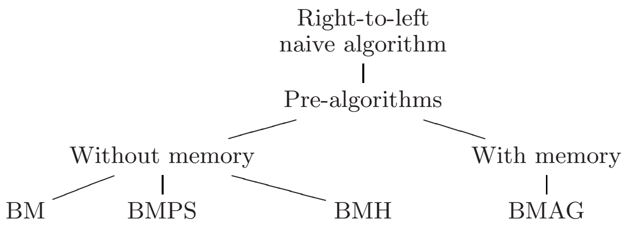


Fig. 3. A taxonomic tree of some right-to-left string-matching algorithms

Acknowledgments

Support for this work came from *Consejo Nacional de Ciencia y Tecnología* (CONACYT) and *Universidad Tecnológica de la Mixteca* (UTM). I thank David Rosenblueth by sharing with me some of his views on program transformation. I am also grateful the reviewers of WFLP'09 for their valuable comments. Thanks Evelina, for your encouragement and great friendship.

References

[AG86] Apostolico, A., Giancarlo, R.: The Boyer–Moore–Galil string searching strategies revisited. SIAM J. Comput. 15(1), 98–105 (1986)

[BdM97] Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall, Englewood Cliffs (1997)

[Bir05] Bird, R.: Polymorphic string matching. In: Haskell 2005, Tallinn, Estonia. Association for Computing Machinery, ACM (September 2005)

[BM77] Boyer, R.S., Strother Moore, J.: A Fast String Searching Algorithm. Communications of the ACM 20(10) (October 1977)

[CD78] Clark, K.L., Darlington, J.: Algorithm classification through synthesis. The Computer Journal 23(1), 61–65 (1978)

[CD89] Consel, C., Danvy, O.: Partial Evaluation of Pattern Matching in Strings. Information Processing Letters 30(2), 79–86 (1989)

[Cla78] Clark, K.L.: Negation as Failure. In: Gallaire, H., Minker, J. (eds.) Logic and Databases, pp. 293–322. Plenum Press, New York (1978)

[CP91] Crochemore, M., Perrin, D.: Two-way string-matching. Journal of the Association for Computing Machinery 38(3) (July 1991)

[CR94] Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press, Oxford (1994)

[CR02] Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing, Singapore (2002)

[Dar78] Darlington, J.: A synthesis of several sorting algorithms. Acta Informatica 11, 1–30 (1978)

[DRK06] Danvy, O., Korsholm, H.R.: On obtaining the boyer–moore string-matching algorithm by partial evaluation. Information Processing Letters 99(4) (August 2006)

[FPP02] Fioravanti, F., Pettorossi, A., Proietti, M.: Specialization with clause splitting for deriving deterministic constraint logic programs. In: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC 2002), Hammamet, Tunisia. IEEE Computer Society Press, Los Alamitos (2002)

[GK94] Gergatsoulis, M., Katzouraki, M.: Unfold/fold Transformations for Definite Clause Programs. In: Hermenegildo, M., Penjam, J. (eds.) PLILP 1994. LNCS, vol. 844. Springer, Heidelberg (1994)

[Hor80] Nigel Horspool, R.: Practical Fast Searching in Strings. Software—Practice and experience 10(6), 501–506 (1980)

[HR01] Hernández, M., Rosenblueth, D.: Development Reuse and the Logic Program Derivation of Two String-Matching Algorithms. In: Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001), September 2001, Florence, Italy, pp. 38–48 (2001)

[HR03] Hernández, M., Rosenblueth, D.: A Disjunctive Partial Deduction of a Right-to-Left String-Matching Algorithm. Information Proceesing Letters 87(5), 235–241 (2003)

[KMP77] Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast Pattern Matching in Strings. SIAM Journal of Computation 6(2), 323–350 (1977)

[Lau89] Lau, K.-K.: A Note on Synthesis and Classification of Sorting Algorithms. Acta Informatica 27, 73–80 (1989)

[Llo87] Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)

[MACD01] Malmkjr, K., Amtoft, T., Consel, C., Danvy, O.: The abstraction and instantiation of string-matching programs. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 332–357. Springer, Heidelberg (2002)

[Par76] Parnas, D.L.: On the design and development of program families. IEEE Transactions of Software Engineering SE-2(1), 1–9 (1976)

[Pep91] Pepper, P.: Literate Program Derivation: A Case Study. In: Broy, M., Wirsing, M. (eds.) CIP 1991. LNCS, vol. 544, pp. 101–124. Springer, Heidelberg (1991)

[PP98] Pettorossi, A., Proietti, M.: Transformation of logic programs. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 697–787. Oxford University Press, Oxford (1998)

[PPR97] Pettorossi, A., Proietti, M., Renault, S.: Enhancing Partial Deduction via Unfold/Fold Rules. In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207. Springer, Heidelberg (1997)

[PS90] Partsch, H.A., Stomp, F.A.: A fast pattern matching algorithm derived by transformational and assertional reasoning. Formal Aspects of Computing 2, 109–122 (1990)

[RS83] Reif, J.H., Scherlis, W.L.: Deriving efficient graphs algorithms. In: Logic of programs (Proceedings 1983). LNCS, vol. 164, pp. 421–441. Springer, Heidelberg (1983)

[SGJ96] Sørensen, M.H., Glück, R., Jones, N.D.: A positive supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)

[Smi91] Smith, D.A.: Partial Evaluation of Pattern Matching in Constraint Logic Programming Language. In: Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 1991, Connecticut, USA, pp. 62–71. ACM Press, New York (1991)

[Ste94] Stephen, G.A.: String Searching Algorithms. Lecture Notes Series on Computing, vol. 3. World Scientific Publishing, Singapore (1994)

[Sun90] Sunday, D.M.: A very fast substring search algorithm. Communications of the ACM 33(8) (August 1990)

[WZ96] Watson, B.W., Zwaan, G.: A taxonomy of sublinear multiple keyword pattern matching algorithms. Science of Computer Programming 27(2), 85–118 (1996)

Type Checking and Inference Are Equivalent in Lambda Calculi with Existential Types

Yuki Kato and Koji Nakazawa

Graduate School of Informatics, Kyoto University, Kyoto 606-8501, Japan
yuki@kuis.kyoto-u.ac.jp, knak@kuis.kyoto-u.ac.jp

Abstract. This paper shows that type-checking and type-inference problems are equivalent in domain-free lambda calculi with existential types, that is, type-checking problem is Turing reducible to type-inference problem and vice versa. In this paper, the equivalence is proved for two variants of domain-free lambda calculi with existential types: one is an implication and existence fragment, and the other is a negation, conjunction and existence fragment. This result gives another proof of undecidability of type inference in the domain-free calculi with existence.

Keywords: undecidability, existential type, type checking, type inference, domain-free type system.

1 Introduction

Existential types correspond to second-order existence in logic by the Curry-Howard isomorphism, and so they are a natural notion from the point of view of logic. They have been also studied actively from the point of view of computer science since Mitchell and Plotkin [11] showed that abstract data types are existential types. Furthermore, calculi with existential types work as suitable target calculi of continuation-passing-style (CPS) translations. Some studies on CPS translations for polymorphic calculi have shown that the negation ($\neg$, which corresponds to continuation types), conjunction ($\wedge$, which corresponds to product types), and existence ($\exists$) fragment of lambda calculus is an essence of a target calculus of CPS translations for various systems, such as the polymorphic lambda calculus [5], the lambda-mu calculus [3,8], and delimited continuations. Hasegawa [9] showed that a $\neg\wedge\exists$-fragment is even more suitable as a target calculus of a CPS translation for delimited continuations such as `shift` and `reset` [2]. These can be seen as an extension of the study of Thielecke [18], in which he showed that the negation and conjunction fragment of a lambda calculus suffices as a target of CPS translations of various first-order calculi.

Domain-free type systems [1], which are in an intermediate style between Church and Curry style, are useful to study some extensions of polymorphic typed calculi and for theoretical studies on CPS translations. In domain-free style lambda calculi, types of parameters of functions are not explicitly annotated in lambda abstraction terms $\lambda x.M$ as in the Curry style, while as in the Church style, terms contain type information for second-order quantifiers, such as a type abstraction $\lambda X.M$ for $\forall$-introduction rule, and a term $\langle A, M\rangle$ with a witness A for $\exists$-introduction rule. In [7], it is shown that an extension of the Damas-Milner polymorphic type assignment system, which can be seen as a Curry-style formulation, with a control operator destroys the type soundness. Similarly, Fujita [3] showed that the Curry-style lambda-mu calculus, which is an extension of the polymorphic lambda calculus and introduced by Parigot [14], does not enjoy the subject reduction property. Fujita introduced a domain-free lambda-mu calculus to have the subject reduction. In addition, the $\neg\wedge\exists$-fragment of the domain-free typed lambda calculus works as a target calculus of a CPS translation for the domain-free lambda-mu calculus.

Some decision problems on typability of terms in typed calculi have been widely studied. One is *type-checking problem* (TC), which is a problem deciding whether $\Gamma \vdash M : A$ is derivable for given Γ, M, and A. *Type-inference problem* (TI) is another problem deciding whether there exist Γ and A such that $\Gamma \vdash M : A$ is derivable for given M. In the usual notation, TC asks $\Gamma \vdash M : A$? for given Γ, M, and A, and TI asks $? \vdash M :?$ for given M. In this paper, TC_0 and TI_0 denote type checking and inference for closed terms, respectively. These questions are fundamentally important in typed lambda calculi.

For polymorphic types, we have already had some results on these problems. Wells [19] showed that TC and TI in the Curry-style polymorphic lambda calculus are equivalent and these problems are undecidable. Two problems are said to be equivalent if one is Turing reducible to the other and vice versa, where a decision problem P is said to be Turing reducible to another problem Q when there exists computable function F such that for each instance p of P, $F(p)$ is an instance of Q which holds if and only if p holds. Nakazawa and Tatsuta [13] showed that TC and TI in the domain-free polymorphic lambda calculus are equivalent, and these are undecidable. On the other hand, despite of their computational importance, properties of existential types have not been studied enough yet. It is only recent that inhabitation problem, which corresponds to provability of formulas, in the $\neg\wedge\exists$-fragment was proved to be decidable in [17]. TC and TI in domain-free lambda calculi with existential types were proved to be undecidable in [12,13]. However any direct relation between TC and TI for existential types has not been known yet.

This paper proves that TC and TI are equivalent in two variants of domain-free lambda calculi with existential types: implication and existence fragment $\mathsf{DF}\text{-}\lambda^{\to\exists}$, and negation, conjunction, and existence fragment $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. Moreover, this result gives another proof of undecidability of TI in $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

First, we prove that TC and TI are equivalent in $\mathsf{DF}\text{-}\lambda^{\to\exists}$. In $\mathsf{DF}\text{-}\lambda^{\to\exists}$, it is easy to prove that TI is Turing reducible to TC. The reduction from TC to TI is proved by adapting the idea of [13]. The key of the proof is the fact that, for given a closed term M and a type A, we can construct another closed term $J_{M,A}$ which is typable if and only if $\vdash M : A$ holds.

Secondly, we prove that TC and TI are equivalent in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. Similarly to $\mathsf{DF}\text{-}\lambda^{\to\exists}$, the proof of the reduction from TC to TI consists of two parts: the reduction from TC to TC_0 and that from TC_0 to TI. However, since $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ does not have implication, we need a non-trivial idea to prove the reduction from TC to TC_0. In this paper, using the well-known fact that the implication $A \to B$ is (classically) equivalent to $\neg(A \wedge \neg B)$, we show that TC can be reduced to TC_0 in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. The proof of the other direction from TI to TC also consists of two parts: TI can be reduced to TI_0 , and TI_0 can be reduced to TC. In order to prove the former part, the above idea can be used.

Figure 1 summarizes the related results including ours. In the diagram, $P \leq Q$ means that the problem P is Turing reducible to Q, and $P \simeq Q$ means that P and Q are equivalent, that is, both $P \leq Q$ and $Q \leq P$ hold. F denotes the polymorphic lambda calculus. SUP means the semi-unification problem and 2UP means the second-order-unification problem. Since undecidability of SUP and 2UP has been already proved by Kfoury et al. [10] and Schubert [15], respectively, all of the problems in the diagram are undecidable. $\simeq^*$ is the main result of this paper, and it gives a new proof of undecidability of TI in $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

$$\begin{array}{lccccc}
\text{Curry style:} & \text{SUP} & \overset{[19]}{\leq} & \text{TC in } F & \overset{[19]}{\simeq} & \text{TI in } F \\[2em]
\text{domain-free style:} & \text{2UP} & \overset{[4]}{\leq} & \text{TC in } \mathsf{DF}\text{-}F & \overset{[13]}{\simeq} & \text{TI in } \mathsf{DF}\text{-}F \\
 & & & \rotatebox{90}{$\leq$}\ [12] & & \rotatebox{90}{$\leq$}\ [12] \\
\text{domain-free style:} & & & \text{TC in } \mathsf{DF}\text{-}\lambda^{\to\exists}/\lambda^{\neg\wedge\exists} & \simeq^* & \text{TI in } \mathsf{DF}\text{-}\lambda^{\to\exists}/\lambda^{\neg\wedge\exists}
\end{array}$$

Fig. 1. TC and TI for polymorphic and existential types

The section 2 introduces the domain-free lambda calculi with existence: $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. We state our main theorems in the section 3, and we prove them in the sections 4 and 5.

2 Domain-Free Lambda Calculi with Existence

In this section, we define the domain-free lambda calculi with Existential types: $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

These calculi are expressive enough to represent every function which is representable in System F, because we can interpret every term of System F in each of $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ by CPS translations [5,8]. Furthermore, as pointed out in [11], the existential types can be seen as the abstract data types in the following sense. If a term M has a type $B[X := A]$, we can hide the information of the type A by constructing the term $\langle A, M\rangle$, which has the existential type $\exists X.B$. A term N of the existential type $\exists X.B$ can be used with $N[Xx.P]$, which intuitively means let $\langle X, x\rangle = N$ in P. In this paper, we use the notation $M[Xx.N]$ following [12,13]. We can write the term $\langle A, M\rangle$ in the style of modules of Standard ML as `struct type` $X = A$ `val` $x = M$ `end`.

Here are examples of terms with the existential type. First, we define a term of the type $\exists X.\neg X \wedge X$.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash F : \neg\texttt{int}\end{array} \quad \dfrac{}{\vdash 1 : \texttt{int}}\,(Ax)}{\vdash \langle F, 1\rangle : \neg\texttt{int} \wedge \texttt{int}}\,(\wedge I)}{\vdash \langle \texttt{int}, \langle F, 1\rangle\rangle : \exists X.\neg X \wedge X}\,(\exists I)$$

We can consider this term $\langle \texttt{int}, \langle F, 1\rangle\rangle$ as a module, which is implemented as $\langle F, 1\rangle$ by the type `int`, but the information of `int` is hidden in the type $\exists X.\neg X \wedge X$. We can give a name to this module by the binding mechanism of λ-calculus such as $(\lambda m.m[Xp.(p\pi_1)(p\pi_2)])\langle \texttt{int}, \langle F, 1\rangle\rangle$. The term $\lambda m.m[Xp.(p\pi_1)(p\pi_2)]$ is typed as follows, where Γ denotes the type assignment $p : \neg X \wedge X$.

$$\dfrac{\dfrac{\dfrac{}{m : \exists X.\neg X \wedge X \vdash m : \exists X.\neg X \wedge X}\,(Ax) \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash p : \neg X \wedge X}\,(Ax)}{\Gamma \vdash p\pi_1 : \neg X}\,(\wedge E_1) \quad \dfrac{\dfrac{}{\Gamma \vdash p : \neg X \wedge X}\,(Ax)}{\Gamma \vdash p\pi_2 : X}\,(\wedge E_2)}{\Gamma \vdash (p\pi_1)(p\pi_2) : \bot}\,(\neg E)}{m : \exists X.\neg X \wedge X \vdash m[Xp.(p\pi_1)(p\pi_2)] : \bot}\,(\exists E)}{\vdash \lambda m.m[Xp.(p\pi_1)(p\pi_2)] : \neg(\exists X.\neg X \wedge X)}\,(\neg I)$$

Users of the module do not need to know which type is used in the implementation of the module. We can consider that the term $(\lambda m.m[Xp.(p\pi_1)(p\pi_2)])\langle \texttt{int}, \langle F, 1\rangle\rangle$ corresponds to the following program of Standard ML.

```
structure m = struct
  type X = int
  val p = (F,1)
end
let val (f,a) = m.p in
  f a
end
```

The signatures of Standard ML corresponds to type annotations to terms of the existential type such as $\langle A, M\rangle^{\exists X.B}$. We have no such annotations in the domain-free style.

2.1 Lambda Calculus with Implication and Existence

First, we define the domain-free lambda calculus DF-$\lambda^{\to\exists}$ with implication ($\to$) and existence ($\exists$).

Definition 1. *The types (denoted by A, B, ... , and called $\to\exists$-types) and the terms (denoted by M, N, ...) of* DF-$\lambda^{\to\exists}$ *are defined by*

$$A ::= X \mid A{\to}A \mid \exists X.A,$$
$$M ::= x \mid \lambda x.M \mid \langle A, M\rangle \mid MM \mid M[Xx.M].$$

X and x denote a type variable and a term variable, respectively. In the type $\exists X.A$, the variable X in A is bound. In the term $\lambda x.M$, the variable x in M is bound. In the term $N[Xx.M]$, the variables X and x in M are bound. A variable is free if it is not bound. A term is closed if it contains no free term variable. We use $\equiv$ to denote syntactic identity modulo renaming of bound variables.

For $n \geq 3$, $A_1 \to \cdots \to A_{n-1} \to A_n$ denotes $A_1 \to (\cdots \to (A_{n-1} \to A_n))$, and $M_1 M_2 \cdots M_n$ denotes $((M_1 M_2) \cdots) M_n$. Γ denotes a finite set of type assignments in the form of $x : A$.

Typing rules of $\mathsf{DF}\text{-}\lambda^{\to\exists}$ *are the following.*

$$\frac{}{\Gamma, x : A \vdash x : A}\ (Ax)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ (\to I) \qquad \frac{\Gamma_1 \vdash M : B \to A \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash MN : A}\ (\to E)$$

$$\frac{\Gamma \vdash N : A[X := B]}{\Gamma \vdash \langle B, N \rangle : \exists X.A}\ (\exists I) \qquad \frac{\Gamma_1 \vdash M : \exists X.A \quad \Gamma_2, x : A \vdash N : C}{\Gamma_1, \Gamma_2 \vdash M[Xx.N] : C}\ (\exists E)$$

In the rule ($\exists E$), Γ_2 and C must not contain X as a free variable.

2.2 Lambda Calculus with Negation, Conjunction, and Existence

Then we define the domain-free lambda calculus $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ with negation ($\neg$), conjunction ($\wedge$), and existence ($\exists$). From the point of view of computation, the negation corresponds to the type of continuations, and the conjunction to the product type.

Definition 2. *The types (denoted by A, B, ... , and called $\neg\wedge\exists$-types) and the terms (denoted by M, N, ...) of* $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ *are defined by*

$$A ::= X \mid \bot \mid \neg A \mid A \wedge A \mid \exists X.A,$$
$$M ::= x \mid \lambda x.M \mid \langle M, M \rangle \mid \langle A, M \rangle \mid MM \mid M\pi_1 \mid M\pi_2 \mid M[Xx.M].$$

Bound and free variables, and closed terms are defined similarly to $\mathsf{DF}\text{-}\lambda^{\to\exists}$. *For $n \geq 3$, $A_1 \wedge \cdots \wedge A_{n-1} \wedge A_n$ denotes $A_1 \wedge (\cdots \wedge (A_{n-1} \wedge A_n))$.*

Typing rules of $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ *are the following.*

$$\frac{}{\Gamma, x : A \vdash x : A}\ (Ax)$$

$$\frac{\Gamma, x : A \vdash M : \bot}{\Gamma \vdash \lambda x.M : \neg A}\ (\neg I) \qquad \frac{\Gamma_1 \vdash M : \neg A \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash MN : \bot}\ (\neg E)$$

$$\frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \langle M, N \rangle : A \wedge B}\ (\wedge I)$$

$$\frac{\Gamma \vdash M : A_1 \wedge A_2}{\Gamma \vdash M\pi_1 : A_1}\ (\wedge E_1) \qquad \frac{\Gamma \vdash M : A_1 \wedge A_2}{\Gamma \vdash M\pi_2 : A_2}\ (\wedge E_2)$$

$$\frac{\Gamma \vdash N : A[X := B]}{\Gamma \vdash \langle B, N \rangle : \exists X.A}\ (\exists I) \qquad \frac{\Gamma_1 \vdash M : \exists X.A \quad \Gamma_2, x : A \vdash N : C}{\Gamma_1, \Gamma_2 \vdash M[Xx.N] : C}\ (\exists E)$$

In the rule ($\exists E$), Γ_2 and C must not contain X as a free variable. Note that the typing rules of $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ *for the terms $\lambda x.M$ and MN differ from those of* $\mathsf{DF}\text{-}\lambda^{\to\exists}$.

3 Type Checking and Type Inference

In this section, we introduce two decision problems on typability of terms, and state our main theorem.

Type checking (TC) is a problem deciding whether $\Gamma \vdash M : A$ is derivable for given Γ, M, and A. *Type inference* (TI) is a problem deciding whether there exist Γ and A such that $\Gamma \vdash M : A$ is derivable for given M. In the usual notation, TC asks $\Gamma \vdash M : A?$ for given Γ, M, and A, and TI asks $? \vdash M :?$ for given M.

These two problems are equivalent in the Curry-style polymorphic lambda calculus [19], and in the domain-free polymorphic lambda calculus [13]. Two problems are said to be equivalent if one is Turing reducible to the other and vice versa. Hence the equivalence of TC and TI means that (i) for any instance $\Gamma \vdash M : A?$ of TC, we can effectively construct a term N such that the answer of the given instance of TC is the same as that of the instance $? \vdash N :?$ of TI, and (ii) for any instance $? \vdash M :?$ of TI, we can effectively construct an instance $\Gamma \vdash N : A?$ of TC whose answer is the same as the given instance of TI. TC_0 and TI_0 denote type checking and type inference for closed terms, respectively.

In general, if a decision problem P_1 is Turing reducible to another decision problem P_2, then decidability of P_2 implies decidability of P_1, and equivalently undecidability of P_1 implies undecidability of P_2. In [19,13], they showed undecidability of TI in the polymorphic lambda calculi by the Turing reducibility of TC to TI. On the other hand, undecidability of TC and TI in the domain-free lambda calculi with existential types has been proved in [12,13] by the reducibility of each problems for polymorphic types to those for existential types. However, direct relationship between TC and TI for existential types has not been known yet. In this paper, we will prove that TC and TI are equivalent in $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

Theorem 1. *1. Type checking and type inference are equivalent in* $\mathsf{DF}\text{-}\lambda^{\to\exists}$*, that is, type checking in* $\mathsf{DF}\text{-}\lambda^{\to\exists}$ *is Turing reducible to type inference in* $\mathsf{DF}\text{-}\lambda^{\to\exists}$ *and vice versa.*

2. Type checking and type inference are equivalent in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$*.*

Since TC is undecidable in these calculi, this result gives another proof of undecidability of TI in them.

For each system, the proof of the reduction from TC to TI consists of two parts. First, we show that TC can be reduced to TC_0. Secondly, we show that TC_0 can be reduced to TI.

The key of the proof of the reduction from TC_0 to TI is the fact that we can effectively construct a term $J_{M,A}$ from a given pair of a closed term M and a type A such that the instance $\vdash M : A$ of TC_0 is equivalent to the instance $\vdash J_{M,A} :?$ of TI. By this fact, we can conclude that TC_0 can be reduced to TI. In order to show that, we borrow the idea of [13] for polymorphic types.

In $\mathsf{DF}\text{-}\lambda^{\to\exists}$, the reduction from TC to TC_0 is easy, whereas the reduction is not easy to prove for $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ due to absence of implication. In our proof, we show that we can construct a $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term $\underline{\lambda}x.M$ for a $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term M and a variable x such that $\Gamma, x : A \vdash M : B$ holds if and only if $\Gamma \vdash \underline{\lambda} x.M : \neg(A \wedge \neg B)$ holds. By this construction, we can prove that TC can be reduced to TC_0 in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

Similarly, the proof of the reduction from TI to TC consists of two parts: the reduction from TI to TI_0, and the reduction from TI_0 to TC. For $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$, the proof of the reduction from TI to TI_0 has the similar difficulties to the case of the reduction from TC to TC_0. We can also use the same technique by $\underline{\lambda} x.M$ to prove it.

We prove the equivalence in $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$ in the section 4, where we show how to construct $J_{M,A}$ from M and A. In the section 5, we show the reduction from problems for open terms to those for closed terms, and prove the equivalence in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

4 TC and TI Are Equivalent in $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$

In this section, we prove that TC and TI are equivalent in $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$.

At first, we show that TI is Turing reducible to TC (that is denoted by TI $\le$ TC).

Proposition 1. *TI in* $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$ *is Turing reducible to TC in* $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$.

Proof. For a given instance $? \vdash M :?$ of TI in $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$, we can effectively construct the list $(x_1, \cdots, x_n)$ of all of the free variables in M. Then the TI problem $? \vdash M :?$ is equivalent to a TC problem $\vdash \lambda y.(\lambda x.y)(\lambda x_1. \cdots \lambda x_n.M) : X{\rightarrow}X?$, where x and y are fresh variables. In fact, if the term $\lambda y.(\lambda x.y)(\lambda x_1. \cdots \lambda x_n.M)$ has the type $X \rightarrow X$, then M has some type. Conversely, if $\Gamma \vdash M : A$ holds for some Γ and A, we have the following for some B,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : X, x : B \vdash y : X}\,(Ax)}{y : X \vdash \lambda x.y : B{\rightarrow}X}\,({\rightarrow}I) \qquad \begin{array}{c}\Gamma \vdash M : A\\ \vdots \\ \vdash \lambda x_1 \cdots \lambda x_n.M : B\end{array}}{y : X \vdash (\lambda x.y)(\lambda x_1 \cdots \lambda x_n.M) : X}\,({\rightarrow}E)}{\vdash \lambda y.(\lambda x.y)(\lambda x_1 \cdots \lambda x_n.M) : X{\rightarrow}X}\,({\rightarrow}I),$$

where we can suppose that $\{x_1, \cdots, x_n\} = \{z \mid (z : C) \in \Gamma\}$ without loss of generality because the left-hand side is the set of all of the free variables of M. $\square$

As we have stated in the previous section, the proof of TC $\le$ TI consists of two steps: TC $\le$ TC_0 and $\mathrm{TC}_0 \le$ TI. It is easy to prove TC $\le$ TC_0 for $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$. In the following, we show $\mathrm{TC}_0 \le$ TI, that is, for a given instance $\vdash M : A?$ of TC_0, we effectively construct a $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$-term $J_{M,A}$ such that the instance $? \vdash J_{M,A} :?$ of TI is equivalent to the given instance of TC_0.

In the rest of this section, O is supposed to be a fixed type variable, and not to be bound by any existential quantifier. $\neg_O A$ denotes $A{\rightarrow}O$.

First, we define some auxiliary functions on types to prove the key lemma.

Definition 3. *1.* $\mathsf{lvar}(A)$ *is the leftmost variable of* A *when it is free in* A*, and otherwise* $\mathsf{lvar}(A)$ *is undefined.* $\mathsf{lvar}(A)$ *is defined by*

$$\begin{aligned}\mathsf{lvar}(X) &\equiv X,\\ \mathsf{lvar}(A\to B) &\equiv \mathsf{lvar}(A),\\ \mathsf{lvar}(\exists X.A) &\equiv \begin{cases}\mathsf{undefined} & (\mathsf{lvar}(A) = X),\\ \mathsf{lvar}(A) & (\text{otherwise}).\end{cases}\end{aligned}$$

2. The left depth $\mathsf{ldep}(A)$ *is the depth from the root to the leftmost variable in the syntax tree of* A*. It does not depend on whether the variable is free or bound.* $\mathsf{ldep}(A)$ *is defined by*

$$\begin{aligned}\mathsf{ldep}(X) &= 0,\\ \mathsf{ldep}(A\to B) &= \mathsf{ldep}(A) + 1,\\ \mathsf{ldep}(\exists X.A) &= \mathsf{ldep}(A) + 1.\end{aligned}$$

3. The left-bound-variable depth $\mathsf{lbdep}(A)$ *is* $\mathsf{ldep}(B)$ *when* A *includes a subexpression* $\exists X.B$ *and the leftmost variable of* A *is bound by this quantifier. When the leftmost variable of* A *is free,* lbdep *is undefined.* $\mathsf{lbdep}(A)$ *is defined by*

$$\begin{aligned}\mathsf{lbdep}(X) &= \mathsf{undefined},\\ \mathsf{lbdep}(A\to B) &= \mathsf{lbdep}(A),\\ \mathsf{lbdep}(\exists X.A) &= \begin{cases}\mathsf{ldep}(A) & (\mathsf{lvar}(A) = X),\\ \mathsf{lbdep}(A) & (\text{otherwise}).\end{cases}\end{aligned}$$

Lemma 1. *1.* $\mathsf{ldep}(A[Y := B]) \neq \mathsf{ldep}(A)$ *implies* $\mathsf{lvar}(A) \equiv Y$*,*
2. $\mathsf{lvar}(A) \equiv X$ *implies* $\mathsf{lbdep}(A[X := B]) = \mathsf{lbdep}(B)$ *and* $\mathsf{lvar}(A[X := B]) \equiv \mathsf{lvar}(B)$*.*

Proof. 1. By induction on A.
When A is $X (X \not\equiv Y)$, $\mathsf{ldep}(X[Y := B]) = \mathsf{ldep}(X)$ holds.
When A is Y, we have $\mathsf{lvar}(Y) \equiv Y$.
When A is $C\to D$, we have

$$\begin{aligned}\mathsf{ldep}((C\to D)[Y := B]) &= \mathsf{ldep}((C[Y := B])\to(D[Y := B]))\\ &= \mathsf{ldep}(C[Y := B]) + 1\end{aligned}$$

and

$$\mathsf{ldep}(C\to D) = \mathsf{ldep}(C) + 1.$$

Therefore, if $\mathsf{ldep}((C\to D)[Y := B]) \neq \mathsf{ldep}(C\to D)$ holds, we have $\mathsf{ldep}(C[Y := B]) \neq \mathsf{ldep}(C)$. From the induction hypothesis, $\mathsf{ldep}(C[Y := B]) \neq \mathsf{ldep}(C)$ implies $\mathsf{lvar}(C) \equiv Y$. Hence, $\mathsf{lvar}(C\to D) \equiv Y$ holds.
When A is $\exists X.C$, we have

$$\mathsf{ldep}((\exists X.C)[Y := B]) = \mathsf{ldep}(C[Y := B]) + 1$$

and

$$\mathsf{ldep}(\exists X.C) = \mathsf{ldep}(C) + 1.$$

Therefore, if $\mathsf{ldep}((\exists X.C)[Y := B]) \neq \mathsf{ldep}(\exists X.C)$ holds, we have $\mathsf{ldep}(C[Y := B]) \neq \mathsf{ldep}(C)$. From the induction hypothesis, $\mathsf{ldep}(C[Y := B]) \neq \mathsf{ldep}(C)$ implies $\mathsf{lvar}(C) \equiv Y$. Hence, $\mathsf{lvar}(\exists X.C) \equiv Y$ holds.

2. By induction on A.

When A is a variable, since we have $\mathsf{lvar}(A) \equiv X$, A is X. Hence, we have $\mathsf{lbdep}(X[X := B]) = \mathsf{lbdep}(B)$ and $\mathsf{lvar}(X[X := B]) \equiv \mathsf{lvar}(B)$.

When A is $C{\to}D$, if $\mathsf{lvar}(C{\to}D) \equiv Y$ holds, we have $\mathsf{lvar}(C) \equiv Y$. From the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lbdep}(C[Y := B]) = \mathsf{lbdep}(B)$, and so we have

$$\begin{aligned}\mathsf{lbdep}((C{\to}D)[Y := B]) &= \mathsf{lbdep}(C[Y := B]{\to}D[Y := B])\\ &= \mathsf{lbdep}(C[Y := B])\\ &= \mathsf{lbdep}(B).\end{aligned}$$

Moreover, from the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B)$, and so we have

$$\begin{aligned}\mathsf{lvar}((C{\to}D)[Y := B]) &\equiv \mathsf{lvar}(C[Y := B]{\to}D[Y := B])\\ &\equiv \mathsf{lvar}(C[Y := B])\\ &\equiv \mathsf{lvar}(B).\end{aligned}$$

When A is $\exists X.C$, if $\mathsf{lvar}(\exists X.C) \equiv Y$ holds, we have $\mathsf{lvar}(C) \equiv Y \not\equiv X$. We can suppose that X is not contained freely in B by renaming the bound variable X. From the induction hypothesis, we have $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B) \not\equiv X$, and then we have

$$\begin{aligned}\mathsf{lbdep}((\exists X.C)[Y := B]) &= \mathsf{lbdep}(\exists X.C[Y := B])\\ &= \mathsf{lbdep}(C[Y := B]).\end{aligned}$$

From the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lbdep}(C[Y := B]) = \mathsf{lbdep}(B)$. Hence we have $\mathsf{lbdep}((\exists X.C)[Y := B]) = \mathsf{lbdep}(B)$. Furthermore, from the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B)$ and so we have

$$\begin{aligned}\mathsf{lvar}((\exists X.C)[Y := B]) &\equiv \mathsf{lvar}(\exists X.C[Y := B])\\ &\equiv \mathsf{lvar}(C[Y := B])\\ &\equiv \mathsf{lvar}(B),\end{aligned}$$

since $\mathsf{lvar}(C[Y := B]) \not\equiv X$ holds. □

By Lemma 1, the following key lemma is proved.

Lemma 2. *If $\Gamma \vdash x\langle \neg_O \exists X.X, x\rangle : A$ is derivable in* $\mathsf{DF}\text{-}\lambda^{\to\exists}$*, then Γ contains $x : \neg_O \exists X.X$.*

Proof. Suppose that $\Gamma \vdash x\langle \neg_O \exists X.X, x\rangle : A$ is derivable, and the type assignment for x in Γ be $x : C_x$.

Since the term $x\langle \neg_O \exists X.X, x\rangle$ is typable, its type derivation is as follows for some C.

$$\dfrac{\dfrac{}{\Gamma \vdash x : \exists Y.C{\to}A}\,(Ax) \qquad \dfrac{\dfrac{}{\Gamma \vdash x : C[X := \neg_O \exists X.X]}\,(Ax)}{\Gamma \vdash \langle \neg_O \exists X.X, x\rangle : \exists Y.C}\,(\exists I)}{\Gamma \vdash x\langle \neg_O \exists X.X, x\rangle : A}\,({\to}E)$$

From the form of (Ax) rules in the derivation, we have

$$\exists Y.C{\to}A \equiv C_x \equiv C[Y := \neg_O \exists X.X].$$

Then we have

$$\begin{aligned}\mathsf{ldep}(C_x) &= \mathsf{ldep}(\exists Y.C{\to}A) = \mathsf{ldep}(\exists Y.C) + 1 = \mathsf{ldep}(C) + 2,\\ \mathsf{ldep}(C_x) &= \mathsf{ldep}(C[Y := \neg_O \exists X.X]).\end{aligned}$$

Hence we have $\mathsf{ldep}(C[Y := \neg_O \exists X.X]) \neq \mathsf{ldep}(C)$, and then $\mathsf{lvar}(C) \equiv Y$ holds by Lemma 1.1. By Lemma 1.2, we have

$$\mathsf{lbdep}(C[Y := \neg_O \exists X.X]) = \mathsf{lbdep}(\neg_O \exists X.X) = \mathsf{lbdep}(\exists X.X) = \mathsf{ldep}(X) = 0.$$

On the other hand, since $\mathsf{lvar}(C) \equiv Y$ holds, we have

$$\mathsf{lbdep}(\exists Y.C{\to}A) = \mathsf{lbdep}(\exists Y.C) = \mathsf{ldep}(C).$$

Therefore, we have $\mathsf{ldep}(C) = 0$, and then C must be a variable. Hence, C is identical to Y because of $\mathsf{lvar}(C) \equiv Y$, and therefore C_x must be $\neg_O \exists X.X$. $\square$

Then we can show the following proposition, from which $\mathrm{TC}_0 \leq \mathrm{TI}$ follows directly.

Proposition 2. *For a closed* $\mathsf{DF}\text{-}\lambda^{\to\exists}$*-term* M *and a* $\to\exists$*-type* A*, we can effectively construct a closed* $\mathsf{DF}\text{-}\lambda^{\to\exists}$*-term* $J_{M,A}$ *such that* $\vdash M : A$ *is derivable if and only if* $\vdash J_{M,A} : B$ *is derivable for some type* B*.*

Proof. Define $J_{M,A}$ as $\lambda x.(\lambda y.x\langle A, M\rangle)(x\langle \neg_O \exists X.X, x\rangle)$, where both x and y are fresh variables. It is easy to see that $\vdash M : A$ implies $\vdash J_{M,A} : \neg_O\neg_O \exists X.X$, because $x : \neg_O \exists X.X \vdash x\langle A, M\rangle : O$ is derivable as follows.

$$\dfrac{\dfrac{}{x : \neg_O \exists X.X \vdash x : \neg_O \exists X.X}\,(Ax) \qquad \dfrac{\begin{array}{c}\vdots\\ \vdash M : A\end{array}}{\vdash \langle A, M\rangle : \exists X.X}\,(\exists I)}{x : \neg_O \exists X.X \vdash x\langle A, M\rangle : O}\,({\to}E)$$

For the converse direction, we use Lemma 2. Suppose that $\vdash J_{M,A} : B$ is derivable for some B. Since $J_{M,A}$ includes $x\langle \neg_O \exists X.X, x\rangle$ as a subterm, the derivation of $\vdash J_{M,A} : B$ includes a derivation of $\Gamma \vdash x\langle\neg_O \exists X.X, x\rangle : B'$ for some Γ and B' as a subderivation. Then Γ contains $x : \neg_O \exists X.X$ by Lemma 2. Because of this, the derivation of $J_{M,A}$ has to include a subderivation of $x : \neg_O \exists X.X \vdash x\langle A, M\rangle : O$ which is the same as the above one. Hence it includes the derivation of $\vdash M : A$. □

Proof (of Theorem 1.1). TI $\leq$ TC is proved in Proposition 1. TC $\leq$ TC$_0$ is easily proved in a similar way to Proposition 1. TC$_0$ $\leq$ TI immediately follows from Proposition 2. □

5 TC and TI Are Equivalent in DF-$\lambda^{\neg\wedge\exists}$

In this section, we prove that TC and TI are equivalent in DF-$\lambda^{\neg\wedge\exists}$.

The proof is similar to DF-$\lambda^{\rightarrow\exists}$, and consists of the following four parts: (i) TC $\leq$ TC$_0$, (ii) TC$_0$ $\leq$ TI, (iii) TI $\leq$ TI$_0$, and (iv) TI$_0$ $\leq$ TC. In contrast to DF-$\lambda^{\rightarrow\exists}$, neither (i) nor (iii) is easy to prove for DF-$\lambda^{\neg\wedge\exists}$ due to absence of implication.

5.1 Translation to Closed Terms

First, we show TC$\leq$TC$_0$ and TI$\leq$TI$_0$. In DF-$\lambda^{\neg\wedge\exists}$, we cannot type the term $\lambda x_1.\cdots\lambda x_n.M$, because N has to be typed with $\bot$ in order to type the lambda abstraction $\lambda x.N$. Therefore, we define a construction $\underline{\lambda}x.M$, which can be considered as an interpretation of the implication introduction in DF-$\lambda^{\neg\wedge\exists}$. It should be noted that the construction can be defined as long as we have negation and conjunction, and so existence is not essential for the discussion in this subsection.

Definition 4. *For any $\neg\wedge\exists$-types A and B, $A \Rightarrow B$ denotes the type $\neg(A \wedge \neg B)$. Similarly to the ordinary implication $\rightarrow$, $A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B$ denotes $A_1 \Rightarrow (\cdots \Rightarrow (A_n \Rightarrow B))$. For a* DF-$\lambda^{\neg\wedge\exists}$*-term M and a variable x, we define $\underline{\lambda}x.M$ as $\lambda c.(\lambda x.(c\pi_2)M)(c\pi_1)$, where c is a fresh term variable.*

It is easy to see that the set of free variables of $\underline{\lambda}x.M$ is the set obtained by removing x from the set of free variables of M.

Lemma 3. *$\Gamma, x : A \vdash M : B$ holds if and only if $\Gamma \vdash \underline{\lambda}x.M : A \Rightarrow B$.*

Proof. Suppose that $\Gamma, x : A \vdash M : B$ holds, and then we have the following type derivation for $\Gamma \vdash \underline{\lambda}x.M : \neg(A \wedge \neg B)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{c : A\wedge\neg B \vdash c : A\wedge\neg B}}{c : A\wedge\neg B \vdash c\pi_2 : \neg B} \quad \begin{matrix}\vdots\\ \Gamma, x : A \vdash M : B\end{matrix}}{\Gamma, c : A\wedge\neg B, x : A \vdash (c\pi_2)M : \bot}}{\Gamma, c : A\wedge\neg B \vdash \lambda x.(c\pi_2)M : \neg A} \quad \dfrac{\dfrac{}{c : A\wedge\neg B \vdash c : A\wedge\neg B}}{c : A\wedge\neg B \vdash c\pi_1 : A}}{\dfrac{\Gamma, c : A\wedge\neg B \vdash (\lambda x.(c\pi_2)M)(c\pi_1) : \bot}{\Gamma \vdash \underline{\lambda}x.M : \neg(A\wedge\neg B)}}$$

Conversely, if we have $\Gamma \vdash \underline{\lambda}x.M : \neg(A \wedge \neg B)$, then its derivation must be in the above form. □

Proposition 3. *1. TC in* $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ *is Turing reducible to* TC_0 *in* $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.
2. TI in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ *is Turing reducible to* TI_0 *in* $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

Proof. 1. For a given instance $x_1 : A_1, \cdots, x_n : A_n \vdash M : B$ of TC, we can effectively construct an instance $\vdash \underline{\lambda} x_1. \cdots \underline{\lambda} x_n.M : A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B$ of TC_0, which is equivalent to the given instance by Lemma 3.

2. For a given $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term M, let the list of free variables of M be $x_1, \cdots, x_n$. It should be noted that we can effectively construct the list. We show that the instance $\vdash \underline{\lambda} x_1. \cdots \underline{\lambda} x_n.M$:? of TI_0 is equivalent to the given instance ? $\vdash M$:? of TI.

If $\Gamma \vdash M : B$ is derivable for some Γ and B, then $\Gamma' \vdash M : B$ is derivable, where Γ' is $\{x_1 : A_1, \cdots, x_n : A_n\}$ and each $x_i : A_i$ is contained in Γ. Then we have $\vdash \underline{\lambda} x_1. \cdots \underline{\lambda} x_n.M : A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B$ by Lemma 3, hence $\underline{\lambda} x_1. \cdots \underline{\lambda} x_n.M$ has a type.

Conversely, if $\underline{\lambda} x_1. \cdots \underline{\lambda} x_n.M$ has a type, then M has some type because it is a subterm of $\underline{\lambda} x_1. \cdots \underline{\lambda} x_n.M$. □

From the point of view of logic, the translation $\underline{\lambda} x.M$ becomes clearer. The judgment $x : A \vdash M : B$ implicitly means the implication $A \rightarrow B$. Since $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ has no implication, we have to interpret the implication by means of negation and conjunction. In order to do that, we use the well-known fact that $A \rightarrow B$ is (classically) equivalent to $\neg(A \wedge \neg B)$, which is denoted by $A \Rightarrow B$ in this paper. Since we cannot conclude B from $A \Rightarrow B$ and A in the intuitionistic logic, $A \Rightarrow B$ is not an intuitionistic implication. We can consider an elimination rule for $\Rightarrow$ such as

$$\frac{\Gamma_1 \vdash M : A \Rightarrow B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M \underline{@} N : \neg\neg B},$$

where $M \underline{@} N$ is defined as $\lambda k.M\langle N, k\rangle$. We can consider that the constructions $\underline{\lambda} x.M$ and $M \underline{@} N$ realize the interpretation of the variant of implication, which is implicitly implemented by "$\vdash$", in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

The translation which maps $\lambda x.M$ to $\underline{\lambda} x.M$ and MN to $M \underline{@} N$ is also important from the point of view of computer science, because it can be considered as a variant of continuation-passing-style translations into the lambda calculus with continuation types and product types. Such translations have been studied in [18,5,8].

In addition, note that we can construct a simpler closed term N and a type C from a given instance $x_1 : A_1, \cdots, x_n : A_n \vdash M : B$ of TC. N and C can be defined as follows:

$$N \equiv \lambda c.(\lambda x_1. \cdots (\lambda x_{n-1}.(\lambda x_n.(c\pi^{n+1}_{n+1})M)(c\pi^{n+1}_n))(c\pi^{n+1}_{n-1}) \cdots)(c\pi^{n+1}_1) : \bot,$$
$$C \equiv \neg(A_1 \wedge \cdots \wedge A_n \wedge \neg B),$$

where π^n_m is the m-th projection for n-tuples, which can be constructed by π_1 and π_2. Then we can show that $x_1 : A_1, \cdots, x_n : A_n \vdash M : B$ holds if and only if $\vdash N : C$ holds. This construction can be used to prove $\mathrm{TI} \leq \mathrm{TI}_0$ as well.

5.2 Proof of Equivalence

We complete the proof of equivalence in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. $\mathrm{TC}_0 \leq \mathrm{TI}$ can be proved similarly to $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$ by replacing $\neg_O$ by $\neg$.

Lemma 4. *If $\Gamma \vdash x\langle\neg\exists X.X, x\rangle : A$ is derivable in $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$, then Γ contains $x : \neg\exists X.X$.*

Proof. The definitions of the auxiliary functions for negation and conjunction are

$$\begin{aligned}\mathsf{lvar}(\neg A) &\equiv \mathsf{lvar}(A),\\ \mathsf{lvar}(A \wedge B) &\equiv \mathsf{lvar}(A),\end{aligned}$$

$$\begin{aligned}\mathsf{ldep}(\neg A) &= \mathsf{ldep}(A) + 1,\\ \mathsf{ldep}(A \wedge B) &= \mathsf{ldep}(A) + 1,\end{aligned}$$

$$\begin{aligned}\mathsf{lbdep}(\neg A) &= \mathsf{lbdep}(A),\\ \mathsf{lbdep}(A \wedge B) &= \mathsf{lbdep}(A),\end{aligned}$$

and the same statement as Lemma 1 holds for $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$. Hence the claim is proved similarly to Lemma 2. $\square$

The following proposition is also proved similarly to $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$.

Proposition 4. *For a closed $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term M and a $\neg\wedge\exists$-type A, we can effectively construct a closed $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$-term $J'_{M,A}$ such that $\vdash M : A$ is derivable if and only if $\vdash J'_{M,A} : B$ is derivable for some type B.*

Proof. Define $J'_{M,A}$ as $\lambda x.(\lambda y.x\langle A, M\rangle)(x\langle\neg\exists X.X, x\rangle)$, where both x and y are fresh variables. The proof is similar to Proposition 2, using Lemma 4. Note that the lambda abstractions and function applications in $J'_{M,A}$ correspond to the introduction and elimination rules of negation. $\square$

The theorem for $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$ is proved as follows.

Proof (of Theorem 1.2). $\mathrm{TC} \leq \mathrm{TC}_0$ and $\mathrm{TI} \leq \mathrm{TI}_0$ are proved by Proposition 3. $\mathrm{TC}_0 \leq \mathrm{TI}$ immediately follows from Proposition 4. $\mathrm{TI}_0 \leq \mathrm{TC}$ is easily proved similarly to $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$, that is, each instance $\vdash M :?$ of TI_0 can be translated to an equivalent instance $\vdash \lambda y.(\lambda x.y)M : \neg\bot$ of TC. $\square$

6 Concluding Remarks

In this paper, we show equivalence between the type checking and the type inference in the domain-free lambda calculi with existential types: $\mathsf{DF}\text{-}\lambda^{\rightarrow\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

As another style for existential types, we can consider a system with no type annotations in terms. The system was introduced in [16], and it has the following rules for existential types.

$$\frac{\Gamma \vdash N : A[X := B]}{\Gamma \vdash \langle \exists, N \rangle : \exists X.A}\ (\exists\mathsf{I}) \qquad \frac{\Gamma_1 \vdash M : \exists X.A \quad \Gamma_2, x : A \vdash N : C}{\Gamma_1, \Gamma_2 \vdash M[x.N] : C}\ (\exists\mathsf{E})$$

Fujita and Schubert [6] call such a style the *type-free style*, and they showed that TC and TI are undecidable in the type-free-style $\lambda^{\rightarrow\exists}$ by the reduction of the second-order unification problem. However, decidability of the problems in the type-free-style $\lambda^{\neg\wedge\exists}$ has not been studied. The direct relations between TC and TI in these type-free-style calculi with existence are not known, either. The technique in this paper essentially use type annotations in terms, and so it cannot be directly adapted to the type-free-style calculi.

The undecidability of the type inference is a negative result for automatic check on safety of program execution. Therefore it is also future work to study on type systems with the existential type in which the type inference problems are decidable. In particular, the type annotations for existential types such as $\langle A, M \rangle^{\exists X.B}$ correspond to the signatures in Standard ML, and so it is important future work to study on the type-related problems in the type systems with such type annotations.

Acknowledgments. We are grateful to Professor Santiago Escobar and the anonymous referees for comments to the previous version of this paper.

References

1. Barthe, G., Sørensen, M.H.: Domain-free pure type systems. Journal of Functional Programming 10, 412–452 (2000)
2. Danvy, O., Fillinski, A.: Representing Control: a Study of the CPS Translation. Mathematical Structures in Computer Science 2(4), 361–391 (1992)
3. Fujita, K.: Explicitly typed $\lambda\mu$-calculus for polymorphism and call-by-value. In: Girard, J.-Y. (ed.) TLCA 1999. LNCS, vol. 1581, pp. 162–177. Springer, Heidelberg (1999)
4. Fujita, K., Schubert, A.: Partially Typed Terms between Church-Style and Curry-Style. In: Watanabe, O., Hagiya, M., Ito, T., van Leeuwen, J., Mosses, P.D. (eds.) TCS 2000. LNCS, vol. 1872, pp. 505–520. Springer, Heidelberg (2000)
5. Fujita, K.: Galois embedding from polymorphic types in to existential types. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 194–208. Springer, Heidelberg (2005)
6. Fujita, K., Schubert, A.: Existential Type Systems with No Types in Terms. In: Curien, P.-L. (ed.) Typed Lambda Calculi and Applications. LNCS, vol. 5608, pp. 112–126. Springer, Heidelberg (2009)
7. Harper, R., Lillibridge, M.: Polymorphic Type Assignment and CPS Conversion. Lisp and Symbolic Computation 6, 361–380 (1993)
8. Hasegawa, M.: Relational parametricity and control. Logical Methods in Computer Science 2(3:3), 1–22 (2006)

9. Hasegawa, M.: (2007) (unpublished manuscript)
10. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: The Undecidability of the Semi-unification Problem. Information and Computation 102, 83–101 (1993)
11. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems 10(3), 470–502 (1988)
12. Nakazawa, K., Tatsuta, M., Kameyama, Y., Nakano, H.: Undecidability of Type-Checking in Domain-Free Typed Lambda-Calculi with Existence. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 478–492. Springer, Heidelberg (2008)
13. Nakazawa, K., Tatsuta, M.: Type Checking and Inference for Polymorphic and Existential Types. In: 15th Computing: the Australasian Theory Symposium (CATS 2009), Conferences in Research and Practice in Information Technology (CRPIT), vol. 94 (2009)
14. Parigot, M.: $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 190–201. Springer, Heidelberg (1992)
15. Schubert, A.: Second-order unification and type inference for Church-style polymorphism. In: 25th Annual ACM Symposium on Principles of Programming Languages (POPL 1998), pp. 279–288 (1998)
16. Tatsuta, M.: Simple saturated sets for disjunction and second-order existential quantification. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 366–380. Springer, Heidelberg (2007)
17. Tatsuta, M., Fujita, K., Hasegawa, R., Nakano, H.: Inhabitance of Existential Types is Decidable in Negation-Product Fragment. In: Proceedings of 2nd International Workshop on Classical Logic and Computation, CLC 2008 (2008)
18. Thielecke, H.: Categorical Structure of Continuation Passing Style. Ph.D. Thesis, University of Edinburgh (1997)
19. Wells, J.B.: Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In: Proceedings of 9th Symposium on Logic in Computer Science (LICS 1994), pp. 176–185 (1994)

Fast and Accurate Strong Termination Analysis with an Application to Partial Evaluation*

Michael Leuschel[1], Salvador Tamarit[2], and Germán Vidal[2]

[1] Institut für Informatik, Universität Düsseldorf, D-40225, Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de
[2] DSIC, Technical University of Valencia, E-46022, Valencia, Spain
{stamarit,gvidal}@dsic.upv.es

Abstract. A logic program strongly terminates if it terminates for any selection rule. Clearly, considering a particular selection rule—like Prolog's leftmost selection rule—allows one to prove more goals terminating. In contrast, a strong termination analysis gives valuable information for those applications in which the selection rule cannot be fixed in advance (e.g., partial evaluation, dynamic selection rules, parallel execution). In this paper, we introduce a fast and accurate size-change analysis that can be used to infer conditions for both strong termination and strong quasi-termination of logic programs. We also provide several ways to increase the accuracy of the analysis without sacrificing scalability. In the experimental evaluation, we show that the new algorithm is up to three orders of magnitude faster than the previous implementation, meaning that we can efficiently deal with programs exceeding 25,000 lines of Prolog.

1 Introduction

Analysing the termination of logic programs is a challenging problem that has attracted a lot of interest (see, e.g., [6,9,26,32] and references therein). However, *strong* termination analysis (i.e., termination for any selection rule) has received little attention, a notable exception being the work by Bezem [3], who introduced the notion of strong termination by defining a sound and complete characterisation (the so-called *recurrent* programs). Also, we can find a well established line of research on termination of logic programs with *dynamic* selection rules (e.g., [28,5,27,30,29]). In these works, however, there are a number of assumptions, like the use of *local* selection rules (a slight extension of the left-to-right selection rule), input-consuming derivations (i.e., derivations where input arguments are not instantiated by SLD resolution steps [4]), etc., which are not useful in our context.

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación (MICINN)* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant GVPRE/2008/001, and by the *UPV* (Programs PAID-05-08 and PAID-06-08). Salvador Tamarit has been partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

In this work, we consider strong (quasi-)termination[1] so that our results can be applied to any application domain where the selection rule is not known in advance or should be dynamically defined, e.g., partial evaluation, resolution with dynamic selection rules, parallel execution, etc.

Consider, for instance, the case of partial evaluation [17], a well-known technique for program specialisation. Within the so-called *offline* approach to partial evaluation, there is a first stage called *binding-time analysis* (BTA) that should analyse the termination of the program and also propagate known data following the program's control flow. In this context, one of the main limitations of previous approaches to the offline partial evaluation of logic programs like, e.g., [8], is that the associated BTA is usually rather expensive and does not scale up well to medium-sized programs. Intuitively speaking, this is mainly due to the fact that the termination analysis and the algorithm for propagating known information are interleaved, so that every time a call is annotated as "not unfoldable", the termination analysis has to be re-executed to take into account that some bindings will not be propagated anymore.

In recent work [20,33], we have shown that this drawback can be overcome by using instead a strong termination analysis based on the size-change principle [18,31]. In this case, both tasks—termination analysis and propagation of known information—are kept independent, so that the termination analysis is done once and for all before the propagation phase, resulting in major efficiency improvements over the previous approach of [8]. Initially, all calls are assumed to be unfolded, but this decision is gradually revised. However, as the size-change analysis is independent of the selection rule, we do not need to recompute this (possibly expensive) phase again after each change.

The new BTA scheme of [20], however, still had some shortcomings concerning both efficiency and accuracy. In particular, the size-change analysis involves computing the composition closure of the so-called *size-change graphs* of the program. This is often an expensive process with a worst case exponential growth factor [18].

In order to overcome this drawback, in this work we introduce an efficient algorithm for the size-change analysis based on the insight that many size change graphs are irrelevant for inferring strong termination and quasi-termination conditions. In particular, we introduce an ordering for size-change graphs, so that only the *weakest* graphs need to be kept without compromising correctness nor accuracy.

Then, we consider the application of the new analysis to the particular domain of offline partial evaluation (cf. Sect. 4) and empirically evaluate the new algorithm. In summary, the empirical results demonstrate the usefulness and scalability of our proposals in practice, meaning that we can efficiently deal with realistic interpreters and systems exceeding 25,000 lines of Prolog.

[1] A computation quasi-terminates if it reaches finitely many different states. This is an essential property in many contexts since it allows one to construct a finite representation of the search space, thus allowing for finite analysis and transformation.

Finally, in Sect. 5 we develop a further improvement of our new algorithm in the context of partial evaluation. Indeed, the fact that the size-change analysis considers *strong* termination may involve a significant loss of accuracy. For instance, given the clauses

$$\begin{array}{l} p(X) \leftarrow q(X,Y), p(Y). \\ q(s(X),X). \end{array}$$

the size-change analysis infers no relation between the sizes of $p(X)$ and $p(Y)$ in the first clause (while, in contrast, one can easily determine that the argument of p decreases from one call to the next one by assuming Prolog's leftmost selection rule). Clearly, this fact makes the size change analysis independent of the selection rule and, particularly, of whether $q(X,Y)$ is unfolded before selecting $p(Y)$ or not. However, in many cases, some partial knowledge is available (e.g., one can safely assume that all *facts* can be unfolded no matter the available information) and could be used to improve the accuracy of the analysis. For this purpose, we develop an extension of the size-change analysis that allows us to propagate some size information from left to right.

2 Fundamentals of Size-Change Analysis

The size-change principle [18] was originally aimed at proving the termination of functional programs. This analysis was adapted to the logic programming setting in [33], where both termination and quasi-termination were analysed. The main difference w.r.t. previous termination analyses for logic programs is that [33] considers *strong* termination, i.e., termination for all computation rules. As mentioned in the introduction, this makes the output of the analysis less accurate but allows the definition of much faster analyses that can be successfully applied in a number of application domains (e.g., for defining a faster binding-time analysis; see [20] for more details).

For conciseness, in the remainder of this paper, we write "(quasi-)termination" to refer to "*strong* (quasi-)termination."

Size-change analysis is based on constructing graphs that represent the decrease of the arguments of a predicate from one call to another. For this purpose, some ordering on terms is required. Analogously to [31], in [33] reduction pairs $(\succsim, \succ)$ consisting of a quasi-order and a compatible well-founded order (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$), both closed under substitutions, were used. The orders $(\succsim, \succ)$ are *induced* from so-called *norms*. Here, we only consider the well-known *term-size* norm $||\cdot||_{ts}$ [11] which counts the number of (non-constant) function symbols. The associated induced orders $(\succsim_{ts}, \succ_{ts})$ are defined as follows: $t_1 \succ_{ts} t_2$ (resp. $t_1 \succsim_{ts} t_2$) if $||t_1\sigma||_{ts} > ||t_2\sigma||_{ts}$ (resp. $||t_1\sigma||_{ts} \geqslant ||t_2\sigma||_{ts}$) for all substitutions σ that make $t_1\sigma$ and $t_2\sigma$ ground. For instance, we have $f(s(X),Y) \succ_{ts} f(X,a)$ since $||f(s(X),Y)\sigma||_{ts} > ||f(X,a)\sigma||_{ts}$ for all σ that makes X and Y ground.

We produce a *size-change graph* $\mathcal{G}$ for every pair (H, B_i) of every clause $H \leftarrow B_1, \ldots, B_n$ of the program. Formally,

Definition 1 (size-change graph). *Let P be a program and $(\succsim, \succ)$ a reduction pair. We define a size-change graph for every clause $p(s_1, \ldots, s_n) \leftarrow Q$ of P and every atom $q(t_1, \ldots, t_m)$ in Q (if any).*

The graph has n output nodes marked with $\{1_p, \ldots, n_p\}$ and m input nodes marked with $\{1_q, \ldots, m_q\}$. If $s_i \succ t_j$ holds, then we have a directed edge from output node i_p to input node j_q marked with $\succ$. Otherwise, if $s_i \succsim t_j$ holds, then we have an edge from output node i_p to input node j_q marked with $\succsim$.

A size-change graph is thus a bipartite labelled graph $\mathcal{G} = (V, W, E)$ where $V = \{1_p, \ldots, n_p\}$ and $W = \{1_q, \ldots, m_q\}$ are the labels of the output and input nodes, respectively, and $E \subseteq V \times W \times \{\succsim, \succ\}$ are the edges.

Example 1. Consider the following program *MLIST*:

$$
\begin{array}{lll}
(c_1) & \mathit{mlist}(L, I, [\,]) \leftarrow \mathit{empty}(L). & \\
(c_2) & \mathit{mlist}(L, I, LI) \leftarrow \mathit{nonempty}(L), \mathit{hd}(L, X), \mathit{tl}(L, R), \mathit{ml}(X, R, I, LI). & \\
(c_3) & \mathit{ml}(X, R, I, [XI|RI]) \leftarrow \mathit{mult}(X, I, XI),\ \mathit{mlist}(R, I, RI). & \\
(c_4) & \mathit{mult}(0, Y, 0). \quad (c_5)\ \mathit{mult}(s(X), Y, Z) \leftarrow \mathit{mult}(X, Y, Z1),\ \mathit{add}(Z1, Y, Z). & \\
(c_6) & \mathit{add}(X, 0, X). \quad (c_7)\ \mathit{add}(X, s(Y), s(Z)) \leftarrow \mathit{add}(X, Y, Z). & \\
(c_8) & \mathit{hd}([X|\_], X). \quad (c_9)\ \mathit{empty}([\,]). & \\
(c_{10}) & \mathit{tl}([\_|R], R). \quad (c_{11})\ \mathit{nonempty}([\_|\_]). &
\end{array}
$$

which is used to multiply all the elements of a list by a given number. The program is somewhat contrived in order to better illustrate our technique.

Here, the size-change graphs associated to, e.g., clause c_3 are as follows:[2]

$$\begin{tikzcd}
1_{ml} \arrow[r, "\succsim_{ts}"] & 1_{mult} \\
2_{ml} & 2_{mult} \\
3_{ml} \arrow[ru, "\succsim_{ts}"] & 3_{mult} \\
4_{ml} \arrow[ru, "\succ_{ts}"] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
1_{ml} & 1_{mlist} \\
2_{ml} \arrow[ru, "\succsim_{ts}"] & 2_{mlist} \\
3_{ml} \arrow[ru, "\succsim_{ts}"] & 3_{mlist} \\
4_{ml} \arrow[ru, "\succ_{ts}"] &
\end{tikzcd}$$

using a reduction pair $(\succsim_{ts}, \succ_{ts})$ induced from the term-size norm.

In order to identify the program *loops*, we should compute roughly the composition closure of the size-change graphs by composing them in all possible ways.

Definition 2 (graph composition, idempotent multigraph). *A multigraph of P is inductively defined to be either a size-change graph of P or the composition (see below) of two multigraphs of P. Given two multigraphs:*

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1) \quad \textit{and} \quad \mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

[2] In general, we denote with p/n a predicate symbol of arity n. However, in the examples, we simply write p for predicate p/n when no confusion can arise.

w.r.t. the same reduction pair $(\succsim, \succ)$*, then the composition*

$$\mathcal{G} \bullet \mathcal{H} = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E)$$

is also a multigraph, where E *contains an edge from* i_p *to* k_r *iff* E_1 *contains an edge from* i_p *to some* j_q *and* E_2 *contains an edge from* j_q *to* k_r*. If some of the edges are labelled with* $\succ$*, then so is the edge in* E*; otherwise, it is labelled with* $\succsim$*.*

We say that a multigraph $\mathcal{G}$ *of* P *is idempotent when* $\mathcal{G} = \mathcal{G} \bullet \mathcal{G}$*. Intuitively speaking, an idempotent multigraph represents a chain of multigraphs.*

Example 2. For the program *MLIST* of Example [1], we have the following four idempotent multigraphs:

$$\begin{tikzcd}
1_{mlist} & 1_{mlist} \\
2_{mlist} \arrow[r, "\succsim_{ts}"] & 2_{mlist} \\
3_{mlist} \arrow[r, "\succ_{ts}"] & 3_{mlist}
\end{tikzcd}
\qquad
\begin{tikzcd}
1_{ml} & 1_{ml} \\
2_{ml} & 2_{ml} \\
3_{ml} \arrow[r, "\succsim_{ts}"] & 3_{ml} \\
4_{ml} \arrow[r, "\succ_{ts}"] & 4_{ml}
\end{tikzcd}
\qquad
\begin{tikzcd}
1_{mult} \arrow[r, "\succ_{ts}"] & 1_{mult} \\
2_{mult} \arrow[r, "\succsim_{ts}"] & 2_{mult} \\
3_{mult} & 3_{mult}
\end{tikzcd}
\qquad
\begin{tikzcd}
1_{add} \arrow[r, "\succsim_{ts}"] & 1_{add} \\
2_{add} \arrow[r, "\succ_{ts}"] & 2_{add} \\
3_{add} \arrow[r, "\succ_{ts}"] & 3_{add}
\end{tikzcd}$$

that represent how the size of the arguments of the four potentially looping predicates changes from one call to another.

The main termination results from [20,33] can be summarised as follows:

– A predicate p/n terminates if every idempotent multigraph for p/n contains at least one edge $i_p \xrightarrow{\succ} i_p$, $1 \leq i \leq n$, such that the i-th argument of every call to this predicate is ground.[3]

– A predicate p/n quasi-terminates if every idempotent multigraph for p/n contains edges $j^1_p \xrightarrow{R_1} 1_p$, $\ldots$, $j^n_p \xrightarrow{R_n} n_p$, $R_i \in \{\succ, \succsim\}$, and the arguments $j^1, \ldots, j^n$ are ground in every call to p/n. Additionally, the considered quasi-order $\succsim$ should be well-founded and *finitely partitioning* [10,32], i.e., there should not be infinitely many "equal" ground terms under $\succsim$.

These conditions, though in principle undecidable, can be approximated in a number of ways. For instance, in the context of partial evaluation, the computed *binding-times*—static for definitely known arguments and dynamic for possibly unknown arguments—can easily be used for this purpose (cf. Sect. [4.1]).

3 A Procedure for Size-Change Analysis

In this section, we introduce a fast and accurate procedure for the size-change analysis of logic programs. In principle, a naive procedure for computing the set of idempotent multigraphs of a program may proceed as follows:

1. First, the size-change graphs of the program are built according to Def. [1].

[3] A more relaxed condition based on the notion of *instantiated enough* w.r.t. a norm [25] can be found in [20].

2. Then, after initialising a set $\mathcal{M}$ with the computed size-change graphs, one proceeds iteratively as follows:
   (a) compute the composition of every pair of (not necessarily different) multigraphs of $\mathcal{M}$;
   (b) update $\mathcal{M}$ with the new multigraphs.

   This process is repeated until no new multigraphs are added to $\mathcal{M}$.

Unfortunately, such a naive algorithm is unacceptably expensive and does not scale up to even simple programs. Therefore, in the following, we introduce a much more efficient procedure. Our algorithm does not compute all size-change graphs, but only a subset of them which is sufficient to produce correct annotations for partial evaluation. Intuitively speaking, it improves the naive procedure by taking into account the following observations:

– Firstly, only the size-change graphs in the path of a (potential) loop need to be constructed. For instance, in Example 1, the size-change graph from *mlist* to *empty* cannot contribute to the construction of any idempotent multigraph. This is a general optimization that is not tied to partial evaluation (similar optimisations can be found, e.g., in [1,12]).
– Secondly, in many cases, computing the idempotent multigraphs for a single predicate for each loop suffices to compute correct annotations for partial evaluation. For instance, in Example 2, the idempotent multigraphs for both *mlist* and *ml* actually refer to the same loop. This is somehow redundant since either the two multigraphs will point out that both predicates terminate or that both of them may loop.
– Finally, when we have multigraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ for a given predicate p/n such that termination of p/n using $\mathcal{G}_1$ always implies termination of p/n using $\mathcal{G}_2$, then we can safely discard $\mathcal{G}_2$. A similar optimisation can be found in [1].

These observations allow us to design a faster procedure for size-change analysis. It proceeds in a stepwise manner as follows:

a) Identifying the program loops. In order to identify the (potential) program loops, we first construct the *call graph* of the program, i.e., a directed graph that contains the predicate symbols as vertices and an edge from predicate p/n to predicate q/m for each clause of the form[4] $p(\overline{t_n}) \leftarrow B_1, \ldots, q(\overline{s_m}), \ldots, B_k$, $k \geq 1$, in the program.

For instance, the call graph of program *MLIST* in Example 1 is as follows:

$$\begin{tikzcd}
& & \mathit{mlist} \arrow[r, bend left] \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & \mathit{ml} \arrow[l, bend left] \arrow[r] & \mathit{mult} \arrow[r] \arrow[loop below] & \mathit{add} \arrow[loop below] \\
\mathit{empty} & \mathit{nonempty} & \mathit{hd} & \mathit{tl} & &
\end{tikzcd}$$

Then, we compute the strongly connected components (SCC) of the call graph and delete both trivial SCCs (i.e., SCCs with a single predicate symbol which is

[4] We use $\overline{t_n}$ to denote the sequence $t_1, \ldots, t_n$.

not self-recursive) and edges between SCCs. We denote the resulting graph with $scc(P)$ for any program P. E.g., for program $MLIST$, $scc(MLIST)$ is as follows:

$$\begin{tikzcd}
mlist \arrow[r, bend left] & ml \arrow[l, bend left] & mult \arrow[loop right] & add \arrow[loop right]
\end{tikzcd}$$

b) Determining the initial set of size-change graphs. We denote by $sc\_graphs(P)$ a subset of the size-change graphs of program P that fulfils the following condition: there is a size-change graph from atom $p(\overline{t_n})$ to atom $q(\overline{s_m})$ in $sc\_graphs(P)$ iff there is an associated edge from p/n to q/m in $scc(P)$. E.g., for program $MLIST$ of Example 1, $sc\_graphs(MLIST)$ contains only four size-change graphs, while the naive approach would have constructed ten size-change graphs.

In principle, only the size-change graphs in $sc\_graphs(P)$ need to be considered in the size-change analysis. This refinement is correct since *idempotent* multigraphs can only be built from the composition of a sequence of size-change graphs that follows the path of a cycle in the call graph (i.e., a path of $scc(P)$).

Furthermore, not all compositions between these size-change graphs are actually required. As mentioned before, computing a single idempotent multigraph for each (potential) program loop suffices. In the following, we say that S is a *cover set* for $scc(P)$ if S contains *at least* one predicate symbol for each loop in $scc(P)$. We denote by $CS(P)$ the set of cover sets for $scc(P)$.

Definition 3 (initial size-change graphs). *Let P be a program and $S \in CS(P)$ be a cover set for $scc(P)$. We denote by $i\_sc\_graphs(P, S)$ the size-change graphs from $sc\_graphs(P)$ that start from a predicate of S.*

Intuitively, the size-change graphs in $i\_sc\_graphs(P, S)$ will act as the *seeds* of our iterative process for computing idempotent multigraphs. As a consequence, only idempotent multigraphs for the predicates of S are produced. Therefore, the termination result of Sect. 2 should be rephrased as follows:

> A predicate p/n terminates if there exists some (not necessarily different) predicate q/m in the same cycle of $scc(P)$ and every idempotent multigraph of q/m contains at least one edge $i_q \xrightarrow{\succ} i_q$, $1 \leq i \leq m$, such that the i-th argument of every call to this predicate q/m is ground. $(*)$

A similar condition could be given for quasi-termination. Proving the correctness of this refinement is not difficult and relies on the fact that either all predicates in a loop are terminating or none.

Example 3. Given the program $MLIST$ of Example 1, we have that both $S_1 = \{mlist/3, mult/3, add/3\}$ and $S_2 = \{ml/4, mult/3, add/3\}$ are cover sets for $scc(MLIST)$. For instance, the set $i\_sc\_graphs(P, S_1)$ contains only the three size-change graphs starting from $mlist/3$, $mult/3$ and $add/3$.

c) Computing the idempotent multigraphs. The core of our improved procedure for size-change analysis is shown in Fig. 1. The algorithm considers the following ordering on multigraphs:

Definition 4 (weaker multigraph). *Given two multigraphs* $\mathcal{G}_1 = \langle V_1, W_1, E_1 \rangle$ *and* $\mathcal{G}_2 = \langle V_2, W_2, E_2 \rangle$*, we say that* $\mathcal{G}_1$ *is weaker than* $\mathcal{G}_2$*, in symbols* $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$*, iff the following conditions hold:*

- *the output and input nodes coincide, i.e.,* $V_1 = V_2$ *and* $W_1 = W_2$*, and*
- *for every edge* $i \xrightarrow{R_1} j \in E_1$*,* $R_1 \in \{\succ, \succsim\}$*, there exists an edge* $i \xrightarrow{R_2} j \in E_2$*,* $R_2 \in \{\succ, \succsim\}$*, such that* $R_1 \sqsubseteq R_2$

where $\succ \sqsubseteq \succ$*,* $\succsim \sqsubseteq \succsim$ *and* $\succsim \sqsubseteq \succ$*, but* $\succ \not\sqsubseteq \succsim$*.*

Basically, if a multigraph $\mathcal{G}$ is weaker than another multigraph $\mathcal{H}$, then we have that whenever termination can be proved with $\mathcal{G}$ only, it could also be proved with both $\mathcal{G}$ and $\mathcal{H}$. Indeed, if $\mathcal{G} \sqsubseteq \mathcal{H}$ and $\mathcal{G}' \sqsubseteq \mathcal{H}'$ then $\mathcal{G} \bullet \mathcal{G}' \sqsubseteq \mathcal{H} \bullet \mathcal{H}'$. Thus, by induction, we can prove that for every size change graph derivable from $\mathcal{H}$ there is a corresponding weaker graph derived from $\mathcal{G}$. Therefore, one can safely discard $\mathcal{H}$ from the computed sets of multigraphs. Intuitively speaking, an idempotent multigraph represents a chain of multigraphs, and this chain is only as strong as its weakest segment.

Example 4. Consider the following four clauses extracted from the regular expression matcher from [21]:

$$\begin{array}{l}
\mathit{generate}(\mathit{or}(X,\_),H,T) \leftarrow \mathit{generate}(X,H,T).\\
\mathit{generate}(\mathit{or}(\_,Y),H,T) \leftarrow \mathit{generate}(Y,H,T).\\
\mathit{generate}(\mathit{star}(\_),T,T).\\
\mathit{generate}(\mathit{star}(X),H,T) \leftarrow \mathit{generate}(X,H,T1), \mathit{generate}(\mathit{star}(X),T1,T).
\end{array}$$

Here, we have the following three size-change graphs:[5]

$$\begin{tikzcd}
1_{gen} \arrow[r, "\succ_{ts}"] & 1_{gen} \\
2_{gen} \arrow[r, "\succsim_{ts}"] & 2_{gen} \\
3_{gen} \arrow[r, "\succsim_{ts}"] & 3_{gen}
\end{tikzcd}
\qquad
\begin{tikzcd}
1_{gen} \arrow[r, "\succ_{ts}"] & 1_{gen} \\
2_{gen} \arrow[r, "\succsim_{ts}"] & 2_{gen} \\
3_{gen} & 3_{gen}
\end{tikzcd}
\qquad
\begin{tikzcd}
1_{gen} \arrow[r, "\succsim_{ts}"] & 1_{gen} \\
2_{gen} & 2_{gen} \\
3_{gen} \arrow[r, "\succsim_{ts}"] & 3_{gen}
\end{tikzcd}$$

using a reduction pair based on the term-size norm, where *generate* is abbreviated to *gen* in the graphs. Here, both the second and third size-change graphs are weaker than the first one, hence the first graph can be safely discarded and also does not have to be composed with other graphs.

The algorithm of Fig. 1 follows these principles:

[5] Note that the first two clauses produce the same size-change graph, otherwise we would have four size-change graphs, one for each body atom in the program.

```
1. **Input:** a program $P$ and a cover set $S \in CS(P)$
2. **Initialisation:**
   $i := 0;\quad \mathcal{M}_i := i\_sc\_graphs(P,S);\quad SC := sc\_graphs(P)$
3. **repeat**
     – $\mathcal{M}_{add} := \emptyset;\ \mathcal{M}_{del} := \emptyset$
     – for all $\mathcal{G}_1 \in \mathcal{M}_i$ and $\mathcal{G}_2 \in SC$ such that $\mathcal{G}_1 \bullet \mathcal{G}_2$ is defined
         (a) $\mathcal{G} := \mathcal{G}_1 \bullet \mathcal{G}_2$
         (b) **if** $\nexists \mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$ such that $\mathcal{G} \sqsubseteq \mathcal{H}$ or $\mathcal{H} \sqsubseteq \mathcal{G}$
             **then** $\mathcal{M}_{add} := \mathcal{M}_{add} \cup \{\mathcal{G}\}$
         (c) **if** $\exists \mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$ such that $\mathcal{G} \sqsubseteq \mathcal{H}$ **then** $\mathcal{M}_{add} := \mathcal{M}_{add} \cup \{\mathcal{G}\}$
             and $\mathcal{M}_{del} := \mathcal{M}_{del} \cup \{\mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}) \mid \mathcal{G} \sqsubseteq \mathcal{H}\}$
     – $\mathcal{M}_{i+1} := (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$
     – $i := i + 1$
   **until** $\mathcal{M}_i = \mathcal{M}_{i+1}$
```

Fig. 1. An improved algorithm for size-change analysis

– In every iteration, we only consider compositions of the form $\mathcal{G}_1 \bullet \mathcal{G}_2$ where $\mathcal{G}_1$ belongs to the current set of multigraphs $\mathcal{M}_i$ and $\mathcal{G}_2$ is one of the original size-change graphs in *sc_graphs*(P).
– Also, those graphs that are stronger than some other graphs are removed from the computed multigraphs in every iteration. Here, $\mathcal{M}_{add}$ denotes the weakest multigraphs that should be added to $\mathcal{M}_i$, while $\mathcal{M}_{del}$ keeps track of the already computed graphs (i.e., from $\mathcal{M}_i \cup \mathcal{M}_{add}$) that should be deleted because a weaker multigraph has been produced.

Example 5. Consider again program *MLIST* of Example 1. By using the improved procedure with the cover set $\{mlist/3, mult/3, add/3\}$, only five compositions are required to get the fixpoint (actually, three of them are only needed to check that a graph is indeed idempotent) and return the final set of idempotent multigraphs (i.e., the first, third and fourth graphs shown in Example 2). With the original algorithm, 48 compositions were required. This is a simple example, but gives an idea of the speedup factor associated to the new algorithm (more details can be found in Sect. 4).

The following result formally states the correctness of keeping only the weakest multigraphs during the iterative process:

Theorem 1. *Let P be a logic program and $\mathcal{M}$ be the set of idempotent multigraphs of P computed using the naive algorithm shown at the beginning of this section. Let $\mathcal{M}'$ be the set of idempotent multigraphs computed with the algorithm of Fig. 1 using a cover set S. Then, a predicate $p/n \in S$ is (quasi-)terminating w.r.t. $\mathcal{M}$ iff it is (quasi-)terminating w.r.t. $\mathcal{M}'$.*

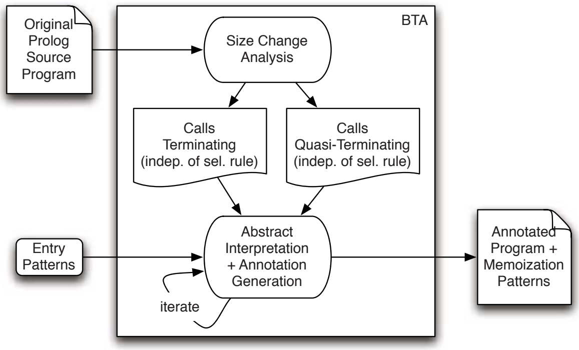

Original Prolog Source Program
BTA
Size Change Analysis
Calls Terminating (indep. of sel. rule)
Calls Quasi-Terminating (indep. of sel. rule)
Entry Patterns
Abstract Interpretation + Annotation Generation
iterate
Annotated Program + Memoization Patterns

Fig. 2. BTA using Size-Change Analysis

As a straightforward corollary, we have that proving termination using the naive algorithm is equivalent to proving termination according to (∗) above using the improved algorithm of Fig. 1 for all program predicates (and not only for those predicates in the cover set).

4 Application to Partial Evaluation and Experiments

In this section, we apply our new algorithm to the case of offline partial evaluation of logic programs, both to show the usefulness of the technique in that setting and also to evaluate its scalability in realistic applications.

4.1 Offline Partial Evaluation of Logic Programs

There are two basic approaches to partial evaluation, differing in the way termination issues are addressed [7,17]. *Online* specializers include a single, monolithic algorithm, while *offline* partial evaluators contain two clearly separated stages: a binding-time analysis (BTA) and the proper partial evaluation. A BTA normally includes both a termination analysis and an algorithm for propagating *static* (i.e., known) information through the program. The output of the BTA is an annotated version of the source program where every call is decorated either with unfold (to be evaluated) or memo (to be residualized, i.e., the call will become part of the residual program); also, every procedure argument is annotated either with static (definitely known at partial evaluation time) or dynamic (possibly unknown at partial evaluation time). Typically, offline partial evaluators are faster but less precise than online partial evaluators.

Figure 2 illustrates the scheme of a BTA that uses the size-change analysis of Fig. 1, where *patterns* are expressions of the form $p(b_1, \ldots, b_n)$, with p/n a predicate symbol of arity n and $b_1, \ldots, b_n$ *binding-times*. Here, we consider a simple domain of binding-times with only two elements: static and dynamic; more refined domains can be found in, e.g., [8].

An offline partial evaluator takes an annotated program and an initial set of atoms and proceeds iteratively as follows:

– First, the initial atoms are unfolded as much as possible according to the program annotations. This is called the *local* level of partial evaluation.
– Then, every atom in the leaves of the incomplete SLD trees produced in the local level are added—after generalising their dynamic arguments—to the set of (to be) partially evaluated atoms. This is called the *global* level of partial evaluation.

Similarly, termination issues can be split into local and global termination, i.e., termination of the local and global levels, respectively. Following the (quasi-) termination results sketched at the end of Sect. 2, source programs are annotated as follows:[6]

Local termination. If all idempotent multigraphs for a predicate p/n include an edge $i_p \xrightarrow{\succ} i_p$ and the i-th argument of p/n is static, then all calls to p/n are annotated with unfold; otherwise, they are annotated with memo.

Global termination. If all idempotent multigraphs for a predicate p/n include an edge $j_p \xrightarrow{R} i_p$ such that $R \in \{\succ, \succsim\}$ and its j-th argument is static, then the i-th argument of p/n can be kept as static (assuming it is known at partial evaluation time); otherwise, it should be annotated as dynamic so that it will be generalised at the global level.

4.2 Prolog Implementation and Empirical Evaluation

We have implemented our new algorithm from Fig. 1 (cf. Sect. 3) for size-change analysis in SICStus Prolog. To be able to measure the effectiveness of the restriction to SCCs (i.e., the restriction to *sc_graphs*(P)) and the restriction to only consider one predicate per loop (i.e., the restriction to *i_sc_graphs*(P, S) for some cover set S), we have provided a way to turn these optimisations off. We also compare to the old implementation from [20], which includes none of the new ideas presented in this paper.

An interesting implementation technique, which all three versions consider (not described in [20]), is the use of hashing[7] to more quickly identify which size-change graphs already exist and which ones can be composed with each other. All these three algorithms are integrated into the same BTA from [20], which provides a command-line interface. The BTA is by default polyvariant (but can be forced to be monovariant) and uses a domain with the following values: static, list_nv (for lists of non-variable terms), list, nv (for non-variable terms), and dynamic. The user can also provide hints to the BTA (see below). The implemented size-change analysis uses a reduction pair induced from the term-size norm.

[6] The groundness of an argument is now replaced by the argument being static.

[7] We note that, in earlier versions of SICStus, `term_hash` generates surprisingly many collisions; a problem which we reported and which has been fixed in version 4.0.5.

Evaluation of Efficiency. Figure 3 contains an overview of our empirical results, where all times are in seconds. A value of 0 means that the timing was below our measuring threshold. The experiments were run on a MacBook Pro with a 2.33 GHz Core2 Duo Processor and 3 GB of RAM. Our BTA was run using SICStus Prolog 4.0.5. The first six benchmarks come from the DPPD [21] library, vanilla, ctl and lambdaint come from [19]. The picemul program is the PIC processor emulator from [14] with 137 clauses and 855 lines of code. javabc and javabc_heap are Java Bytecode interpreters from [13] with roughly 100 clauses. peval includes over 2500 lines of Prolog from a partial evaluator for the ground representation from [23]. self_app are the 1925 lines of our size-change analysis and BTA itself. dSL is an interpreter of 444 lines for the dSL specification language [34]. csp is the core interpreter for full CSP-M from [24], consisting of 1771 lines of code. prob is the core interpreter of ProB [22] for B machines, not containing the kernel predicates or the model checker. It consists of 1910 lines of code and deals with B expressions, predicates and substitutions. promela is an interpreter for the full Promela language (see, e.g., [16]), consisting of 1148 lines of code. Finally, goedel is the source code of the Gödel system [15] consisting of 27354 lines of Prolog.[8] The "noentry" annotation in Fig. 3 means that no entry point was provided, hence only the size-change analysis was performed (and no propagation of static information).

The output of the new BTA (without SCC) and the old BTA from [20] are identical as far as local and global annotations are concerned.

In summary, the new size change analysis is always faster and we see improvements of roughly three orders of magnitude on the most complicated examples (up to a factor of 3500 for prob (noentry)). We are able to deal with realistic interpreters and systems exceeding 25K lines of code. For goedel, a small part of the inferred termination conditions are as follows:

```
is_not_terminating(parse_language1, 6, [d,_,_,_,_,_]).
global_binding_times(parse_language1, 6, [s,d,s,s,d,s]).
is_not_terminating(build_delay_condition, 4, [d,d,_,_]).
global_binding_times(build_delay_condition, 4, [s,s,d,d]).
```

In particular, this means that the analysis has been able to infer that the predicate `parse_language1` can be unfolded if the first argument is static, and that the first, third, fourth and last argument do not need to be generalised to ensure quasi-termination.

Compared to the BTA from [8] using binary clauses rather than size-change analysis, the difference is even more striking. This BTA is in turn, e.g., 200 times slower than the old BTA for the picemul example; see [20]. We have also tried the latest version of Terminweb,[9] based upon [6]. However, the online version failed to terminate successfully on, e.g., the picemul example (for which our

[8] Downloaded from `http://www.cs.bris.ac.uk/Research/LanguagesArchitecture/goedel/` and put into a single file, removing module declarations and adapting some of the code for SICStus 4.

[9] `http://www.cs.bgu.ac.il/~mcodish/TerminWeb/`

Benchmark	Old BTA from [20]	New BTA (without SCC)	New BTA (with SCC)
contains.kmp	0.01	0.00	0.00
imperative-power	2.35	0.03	0.02
liftsolve.app	0.02	0.01	0.01
match-kmp	0.00	0.00	0.00
regexp.r3	0.01	0.00	0.00
ssuply	0.01	0.01	0.01
vanilla	0.01	0.00	0.00
lambdaint	0.17	0.02	0.02
picemul	0.31	0.15	0.15
picemul (noentry)	0.18	0.01	0.01
ctl	0.03	0.02	0.02
javabc	0.03	0.03	0.03
javabc_heap	0.09	0.09	0.09
peval	0.48	0.15	0.06
self_app (noentry)	0.34	0.20	0.05
dSL	0.03	0.01	0.01
csp (noentry)	5.16	0.21	0.09
prob	387.12	1.41	0.61
prob (noentry)	386.63	0.79	0.11
promela (noentry)	330.05	0.35	0.34
goedel (noentry)	1750.90	13.32	2.61

Fig. 3. Empirical results (times in milliseconds)

size-change analysis takes 0.01 s). We have also tried TermiLog,[10] but it timed out after 4 minutes (the maximum time that can be set in the online version).

Evaluation of Precision. Without the use of the proposed optimisations in the algorithm of Fig. 1, the precision remains unchanged w.r.t. [20], and as such the same specialisations can be achieved as described in [20] using hints: e.g., Jones-optimal specialisation for vanilla, reproducing the decompilation from Java bytecode to CLP from [13] or automatically generating the generated code from [14] for picemul.

With the SCC optimisations, we reduce the number of predicates that are memoised. This in turn also reduces the number of hints that a user has to provide to obtain the desired specialisation.

For example, the vanilla example required two hints in [20] and now only one hint is required to obtain a good specialisation. For lambdaint 6 hints were required in [20] to get good performance. Now only two hints are required, expressing the fact that the expression being evaluated and the list of bound variable names are expected to be static and should not be generalised away by the BTA.[11] In the following section we show how the precision of the size-change

[10] `http://www.cs.huji.ac.il/~naomil/termilog.php`

[11] This does not give exactly the same result; the solution with 6 hints memoises on `eval_if`, which in this case leads to a more efficient version than memoising on `eval`.

analysis can be further improved in the setting of partial evaluation, further reducing the need for hints.

5 Propagating Partial Left-to-Right Information

In this section, we extend the size-change analysis in order to right-propagate size information in some cases. Consider, e.g., clause (c_2) in Example 1:

$$(c_2) \quad \mathit{mlist}(L, I, LI) \leftarrow \mathit{nonempty}(L), \mathit{hd}(L, X), \mathit{tl}(L, R), \mathit{ml}(X, R, I, LI).$$

Since our size-change analysis considers *strong* termination, we compare the size of the head of the clause with the size of each atom in the body independently. Therefore, we get no relation between the sizes of list L in the head and its head X and tail R in the call to *ml*.

In some cases, however, one might assume some additional restrictions. For instance, in many partial evaluators a left-to-right selection rule is used with the only exception that those calls which are annotated with `memo` are skipped. Therefore, if we know that some calls can be fully unfolded without entering an infinite loop (the case, e.g., of non-recursive predicates), then one can safely propagate the size relationships for the success patterns of these calls to the subsequent atoms in the clause. In principle, these "fully unfoldable" calls can be detected using a standard left-termination analysis (i.e., one that considers a standard left-to-right computation rule), e.g., [6], while size relations of success patterns can be obtained from the computation of the convex hull of [2]. Here, though, we consider that this information is provided by the user by means of hints of the form `'$FULLYUNFOLD'(p,n,size_relations)` where `size_relations` are the interargument size relations for the success patterns of `p/n`. For instance, for the program *MLIST* of Ex. 1, we may have the following hints:

```
'$FULLYUNFOLD'(hd,2,[1>2]).      '$FULLYUNFOLD'(tl,2,[1>2]).
```

which should be read as "when the call to *hd* (resp. *tl*) succeeds, the size of its first argument is strictly greater than the size of its second argument". We note that, in order to be safe, the interargument size relations should be based on the same norm used to induce the reduction pair considered in the construction of the size-change graphs.

Let us now describe how the size-change analysis can be improved by using this new kind of hints. Consider a clause of the form

$$P \leftarrow Q_1, \ldots, Q_{i-1}, p(t_1, \ldots, t_n), Q_{i+1}, \ldots, Q_m.$$

together with the hint `'$FULLYUNFOLD'(p,n,I)`. Then, we first replace this clause by the following ones:

$$\begin{array}{l} P \leftarrow Q_1, \ldots, Q_{i-1}, p_{entry}(x_1, \ldots, x_k, t_1, \ldots, t_n). \\ p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n) \leftarrow p(y_1, \ldots, y_n), p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n). \\ p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n) \leftarrow Q_{i+1}, \ldots, Q_m. \end{array}$$

where

$$\{x_1, \ldots, x_k\} = (\mathcal{V}ar(P, Q_1, \ldots, Q_{i-1}) \cap \mathcal{V}ar(Q_{i+1}, \ldots, Q_m)) \setminus \mathcal{V}ar(p(t_1, \ldots, t_n))$$

This transformation is clearly safe w.r.t. SLD resolution since the original clause can be obtained by just unfolding both p_{entry} and p_{exit}.

Now, the size-change graphs of the first and third clauses are computed as usual. For the second clause, however, we assume that the atom $p(y_1, \ldots, y_n)$ could be fully unfolded producing the set of clauses

$$\begin{array}{l} p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_1 \leftarrow p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_1. \\ \ldots \\ p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_j \leftarrow p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_j. \end{array}$$

where $\sigma_1, \ldots, \sigma_j$ are the computed answers and the set of interargument size relations I safely approximates the size relations between the arguments of p_{entry} and p_{exit}. Note that we do not need to fully unfold p/n to construct the size-change graphs (it is rather a device to show the correctness of our approach). Formally, for every relation $i > j$ (resp. $i \geqslant j$) in the interargument size relations for p/n, we should add an edge $i_{p_{entry}} \xrightarrow{\succ} j_{p_{exit}}$ (resp. $i_{p_{entry}} \xrightarrow{\succsim} j_{p_{exit}}$) to the size-change graph from p_{entry} to p_{exit}. Moreover, we add an edge of the form $i_{p_{entry}} \xrightarrow{\succsim} i_{p_{exit}}$ since both p_{entry} and p_{exit} are actually the same predicate.

For instance, by considering the previous hints for program $MLIST$, the clause (c_2) is transformed into

$$\begin{array}{ll} (c_{21}) & \mathit{mlist}(L, I, LI) \leftarrow \mathit{nonempty}(L), \mathit{hd}_{entry}(L, X, I, LI). \\ (c_{22}) & \mathit{hd}_{entry}(L, X, I, LI) \leftarrow \mathit{hd}(L, X), \mathit{hd}_{exit}(L, X, I, LI). \\ (c_{23}) & \mathit{hd}_{exit}(L, X, I, LI) \leftarrow \mathit{tl}_{entry}(L, R, X, I, LI). \\ (c_{24}) & \mathit{tl}_{entry}(L, R, X, I, LI) \leftarrow \mathit{tl}(L, R), \mathit{tl}_{exit}(L, R, X, I, LI). \\ (c_{25}) & \mathit{tl}_{exit}(L, R, X, I, LI) \leftarrow \mathit{ml}(X, R, I, LI). \end{array}$$

Now, by using the interargument size relations for hd and tl, we construct the following size-change graphs associated to clauses c_{22} and c_{24}:

$$\begin{tikzcd} 1_{hd_{entry}} \arrow[r, "\succsim_{ts}"] \arrow[dr, "\succ_{ts}"] & 1_{hd_{exit}} \\ 2_{hd_{entry}} \arrow[r, "\succsim_{ts}"] & 2_{hd_{exit}} \\ 3_{hd_{entry}} \arrow[r, "\succsim_{ts}"] & 3_{hd_{exit}} \\ 4_{hd_{entry}} \arrow[r, "\succsim_{ts}"] & 4_{hd_{exit}} \end{tikzcd}
\qquad
\begin{tikzcd} 1_{tl_{entry}} \arrow[r, "\succsim_{ts}"] \arrow[dr, "\succ_{ts}"] & 1_{tl_{exit}} \\ 2_{tl_{entry}} \arrow[r, "\succsim_{ts}"] & 2_{tl_{exit}} \\ 3_{tl_{entry}} \arrow[r, "\succsim_{ts}"] & 3_{tl_{exit}} \\ 4_{tl_{entry}} \arrow[r, "\succsim_{ts}"] & 4_{tl_{exit}} \\ 5_{tl_{entry}} \arrow[r, "\succsim_{ts}"] & 5_{tl_{exit}} \end{tikzcd}$$

Finally, by constructing the size-change graphs for clauses c_{21}, c_{23} and c_{25} as usual, the size-change analysis is now able to infer the right relation between the sizes of list L in the atom $\mathit{mlist}(L, I, LI)$ and the head X and tail R in the atom $\mathit{ml}(X, R, I, LI)$.

6 Discussion and Conclusion

In this paper, we have presented a new algorithm to perform strong termination and quasi-termination inference using size-change analysis. The experiments have shown that we can analyse the full 25K lines of source code of the Gödel system in under three seconds. The main application of this algorithm is for offline partial evaluation of large programs. In the experimental evaluation we have shown that, with our new algorithm, we can now deal with realistic interpreters, such as the interpreter for the full B specification language from [22]. Together with the selective use of hints [20], we have obtained both a scalable and an effective partial evaluation procedure. The logical next step is to bring this work to practical fruition, by, e.g., optimising the interpreter from [22] for particular specifications, speeding up the animation and model checking process. This challenge has been on our research agenda for quite a while, and we now believe that the goal can be achieved in the near future. One remaining technical hurdle is the treatment of `meta_predicate` annotations (the B interpreter uses meta-predicates to implement delaying versions of negation and findall).

References

1. Ben-Amram, A.M., Lee, C.S.: Program termination analysis in polynomial time. ACM TOPLAS 29(1) (2007)
2. Benoy, F., King, A., Mesnard, F.: Computing convex hulls with a linear solver. TPLP 5(1-2), 259–271 (2005)
3. Bezem, M.: Strong Termination of Logic Programs. Journal of Logic Programming 15(1&2), 79–97 (1993)
4. Bossi, A., Etalle, S., Rossi, S.: Properties of input-consuming derivations. TPLP 2(2), 125–154 (2002)
5. Bossi, A., Etalle, S., Rossi, S., Smaus, J.-G.: Termination of simply moded logic programs with dynamic scheduling. ACM Trans. Comput. Log. 5(3), 470–507 (2004)
6. Codish, M., Taboch, C.: A semantic basis for the termination analysis of logic programs. Journal of Logic Programming 41(1), 103–123 (1999)
7. Consel, C., Danvy, O.: Tutorial notes on Partial Evaluation. In: Proc. of POPL 1993, pp. 493–501. ACM Press, New York (1993)
8. Craig, S.-J., Gallagher, J., Leuschel, M., Henriksen, K.S.: Fully Automatic Binding-Time Analysis for Prolog. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 53–68. Springer, Heidelberg (2005)
9. De Schreye, D., Decorte, S.: Termination of logic programs: The never ending story. The Journal of Logic Programming 19 & 20, 199–260 (1994)
10. Decorte, S., De Schreye, D., Leuschel, M., Martens, B., Sagonas, K.F.: Termination Analysis for Tabled Logic Programming. In: Fuchs, N.E. (ed.) LOPSTR 1997. LNCS, vol. 1463, pp. 111–127. Springer, Heidelberg (1998)
11. Van Gelder, A.: Deriving constraints among argument sizes in logic programs. Ann. Math. Artif. Intell. 3(2-4), 361–392 (1991)
12. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and Improving Dependency Pairs. J. Autom. Reasoning 37(3), 155–203 (2006)
13. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Improving the decompilation of Java bytecode to Prolog by partial evaluation. Electr. Notes Theor. Comput. Sci. 190(1), 85–101 (2007)

14. Henriksen, K.S., Gallagher, J.: Abstract interpretation of pic programs through logic programming. In: SCAM, pp. 184–196. IEEE Computer Society, Los Alamitos (2006)
15. Hill, P., Lloyd, J.W.: The Gödel Programming Language. MIT Press, Cambridge (1994)
16. Holzmann, G.J.: The model checker Spin. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
17. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Englewood Cliffs (1993)
18. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The Size-Change Principle for Program Termination. SIGPLAN Notices (Proc. of POPL 2001) 28, 81–92 (2001)
19. Leuschel, M., Craig, S.-J., Bruynooghe, M., Vanhoof, W.: Specialising Interpreters Using Offline Partial Deduction. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 340–375. Springer, Heidelberg (2004)
20. Leuschel, M., Vidal, G.: Fast Offline Partial Evaluation of Large Logic Programs. In: Hanus, M. (ed.) Logic-Based Program Synthesis and Transformation. LNCS, vol. 5438, pp. 119–134. Springer, Heidelberg (2009)
21. Leuschel, M.: The ecce partial deduction system and the dppd library of benchmarks. Obtainable via 1996-2002, http://www.ecs.soton.ac.uk/~mal
22. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
23. Leuschel, M., De Schreye, D.: Creating specialised integrity checks through partial evaluation of meta-interpreters. The Journal of Logic Programming 36(2), 149–193 (1998)
24. Leuschel, M., Fontaine, M.: Probing the depths of CSP-M: A new FDR-compliant validation tool. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 278–297. Springer, Heidelberg (2008)
25. Lindenstrauss, N., Sagiv, Y.: Automatic Termination Analysis of Logic Programs. In: Proc. of ICLP 1997, pp. 63–77. MIT Press, Cambridge (1997)
26. 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)
27. Marchiori, E., Teusink, F.: Termination of Logic Programs with Delay Declarations. J. Log. Program. 39(1-3), 95–124 (1999)
28. Naish, L.: Coroutining and the construction of terminating logic programs. Australian Computer Science Communications 15(1), 181–190 (1993)
29. Smaus, J.-G.: Termination of Logic Programs Using Various Dynamic Selection Rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 43–57. Springer, Heidelberg (2004)
30. Smaus, J.-G., Hill, P.M., King, A.: Verifying termination and error-freedom of logic programs with block declarations. TPLP 1(4), 447–486 (2001)
31. Thiemann, R., Giesl, J.: The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. Applicable Algebra in Engineering, Communication and Computing 16(4), 229–270 (2005)
32. Verbaeten, S., Sagonas, K., De Schreye, D.: Termination Proofs for Logic Programs with Tabling. ACM Transactions on Computational Logic 2(1), 57–92 (2001)
33. Vidal, G.: Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In: Proc. of PEPM 2007, pp. 51–60. ACM Press, New York (2007)
34. De Wachter, B., Genon, A., Massart, T., Meuter, C.: The formal design of distributed controllers with $_d$sl and Spin. Formal Asp. Comput. 17(2), 177–200 (2005)

New Results on Type Systems for Functional Logic Programming[⋆]

Francisco J. López-Fraguas, Enrique Martin-Martin, and Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, emartinm@fdi.ucm.es, juanrh@fdi.ucm.es

Abstract. Type systems are widely used in programming languages as a powerful tool providing safety to programs, and forcing the programmers to write code in a clearer way. Functional logic languages have inherited Damas & Milner type system from their functional part due to its simplicity and popularity. In this paper we address a couple of aspects that can be subject of improvement. One is related to a problematic feature of functional logic languages not taken under consideration by standard systems: it is known that the use of *opaque* HO patterns in left-hand sides of program rules may produce undesirable effects from the point of view of types. We re-examine the problem, and propose a Damas & Milner-like type system where certain uses of HO patterns (even opaque) are permitted while preserving type safety, as proved by a subject reduction result that uses *HO-let-rewriting*, a recently proposed reduction mechanism for HO functional logic programs. The other aspect is the different ways in which polymorphism of local definitions can be handled. At the same time that we formalize the type system, we have made the effort of technically clarifying the overall process of type inference in a whole program.

1 Introduction

Type systems for programming languages are an active area of research [18], no matter which paradigm one considers. In the case of functional programming, most type systems have arisen as extensions of Damas & Milner's [4], for its remarkable simplicity and good properties (decidability, existence of principal types, possibility of type inference). Functional logic languages [12,8,7], in their practical side, have inherited more or less directly Damas & Milner's types. In principle, most of the type extensions proposed for functional programming could be also incorporated to functional logic languages (this has been done, for instance, for type classes in [15]). However, if types are not only decoration but are to provide safety, one should be sure that the adopted system has indeed good

⋆ This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03), STAMP (TIN2008-06622-C03-01), Promesas-CAM (S-0505/TIC/0407) and GPD-UCM (UCM-BSCH-GR58/08-910502).

properties. In this paper we tackle a couple of orthogonal aspects of existing FLP systems that are problematic or not well covered by standard Damas & Milner systems. One is the presence of so called *HO patterns* in programs, an expressive feature allowed in some systems and for which a sensible semantics exists [5]; however, it is known that unrestricted use of HO patterns leads to type unsafety, as recalled below. The second is the degree of polymorphism assumed for local pattern bindings, a matter with respect to which existing FP or FLP systems vary greatly.

The rest of the paper is organized as follows. The next two subsections further discuss the two mentioned aspects. Sect. 2 contains some preliminaries about FL programs and types. In Sect. 3 we expose the type system and prove its soundness wrt. the *let rewriting* semantics of [11]. Sect. 4 contains a type inference relation, which let us find the most general type of expressions. Sect. 5 presents a method to infer types for programs. Finally, Sect. 6 contains some conclusions and future work. Omitted proofs can be found in [13].

1.1 Higher Order Patterns

In our formalism patterns appear in the left-hand side of rules and in lambda or let expressions. Some of these patterns can be HO patterns, if they contain partial applications of function or constructor symbols. HO patterns can be a source of problems from the point of view of the types. In particular, it was shown in [6] that unrestricted use of HO patterns leads to loss of *subject reduction*, an essential property for a type system expressing that evaluation does not change types. The following is a crisp example of the problem.

Example 1 (Polymorphic Casting [2]). Consider the program consisting of the rules $snd\ X\ Y \rightarrow Y$, $and\ true\ X \rightarrow X$, $and\ false\ X \rightarrow false$, with the usual types inferred by a classical Damas & Milner algorithm. Then we can write the functions $unpack\ (snd\ X) \rightarrow X$ and $cast\ X \rightarrow unpack\ (snd\ X)$, whose inferred types will be $\forall\alpha.\forall\beta.(\alpha \rightarrow \alpha) \rightarrow \beta$ and $\forall\alpha.\forall\beta.\alpha \rightarrow \beta$ respectively. It is clear that the expression $and\ (cast\ 0)\ true$ is well-typed, because $cast\ 0$ has type *bool* (in fact it has any type), but if we reduce that expression using the rules of *cast* and *unpack* the resulting expression $and\ 0\ true$ is ill-typed.

The problem arises when dealing with HO patterns, because unlike FO patterns, knowing the type of a HO pattern does not always permit us to know the type of its subpatterns. In the previous example the cause is function *co*, because its pattern $snd\ X$ is *opaque* and shadows the type of its subpattern X. Usual inference algorithms treat this opacity as polymorphism, and that is the reason why it is inferred a completely polymorphic type for the result of the function *co*.

In [6] the appearance of any opaque pattern in the left-hand side of the rules is prohibited, but we will see that it is possible to be less restrictive. The key is making a distinction between **transparent** and **opaque** variables of a pattern: a variable is transparent if its type is univocally fixed by the type of the pattern, and is opaque otherwise. We call a variable of a pattern **critical** if it is opaque in the pattern and also appears elsewhere in the expression. The formal definition of opaque and critical variables will be given in Sect. 3. With these notions we can relax the situation in [6], prohibiting only those patterns having critical variables.

1.2 Local Definitions

Functional and functional logic languages provide syntax to introduce local definitions inside an expression. But in spite of the popularity of let-expressions, different implementations treat them differently because of the polymorphism they give to bound variables. This difference can be observed in Ex. 2, being $(e_1, \ldots, e_n)$ and $[e_1, \ldots, e_n]$ the usual tuple and list notation respectively.

Example 2 (let expressions). Let e_1 be $let\ F = id\ in\ (F\ true, F\ 0)$, and e_2 be $let\ [F, G] = [id, id]\ in\ (F\ true, F\ 0, G\ 0, G\ false)$

Intuitively, e_1 gives a new name to the identity function and uses it twice with arguments of different types. Surprisingly, not all implementations consider this expression as well-typed, and the reason is that F is used with different types in each appearance: $bool \to bool$ and $int \to int$. Some implementations as Clean 2.2, PAKCS 1.9.1 or KICS 0.81893 consider that a variable bound by a let-expression must be used with the same type in all the appearances in the body of the expression. In this situation we say that lets are completely monomorphic, and write let_m for it.

On the other hand, we can consider that all the variables bound by the let-expression may have different but coherent types, i.e., are treated polymorphically. Then expressions like e_1 or e_2 would be well-typed. This is the decision adopted by Hugs Sept. 2006, OCaml 3.10.2 or F# Sept. 2008. In this case, we will say that lets are completely polymorphic, and write let_p.

Finally, we can treat the bound variables monomorphically or polymorphically depending on the form of the pattern. If the pattern is a variable, the let

Programming language and version	let_m	let_{pm}	let_p
GHC 6.8.2		×	
Hugs Sept. 2006			×
Standard ML of New Jersey 110.67		×	
Ocaml 3.10.2			×
F# Sept. 2008			×
Clean 2.0	×		
$\mathcal{TOY}$ **2.3.1***	×		
Curry PAKCS 1.9.1	×		
Curry Münster 0.9.11		×	
KICS 0.81893	×		

(*) we use `where` instead of `let`, not supported by $\mathcal{TOY}$

Fig. 1. Let expressions in different programming languages

treats it polymorphically, but if it is compound the let treats all the variables monomorphically. This is the case of GHC 6.8.2, SML of New Jersey v110.67 or Curry Münster 0.9.11. In this implementations e_1 is well-typed, while e_2 not. We call this kind of let-expression let_{pm}.

Fig. 1 summarizes the decisions of various implementations of functional and functional logic languages. The exact behavior wrt. types of local definitions is usually not well documented, not to say formalized, in those systems. One of our contributions is this paper is to technically clarify this question by adopting a neutral position, and formalizing the different possibilities for the polymorphism of local definitions.

2 Preliminaries

We assume a signature $\Sigma = DC \cup FS$, where DC and FS are two disjoint sets of *data constructor* and *function* symbols resp., all them with associated arity. We write DC^n (resp FS^n) for the set of constructor (function) symbols of arity n. We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. We define the set of *patterns* $Pat \ni t ::= X \mid c\ t_1 \ldots t_n\ (n \leq k) \mid f\ t_1 \ldots t_n\ (n < k)$, where $c \in DC^k$ and $f \in FS^k$; and the set of *expressions* $Exp \ni e ::= X \mid c \mid f \mid e_1\ e_2 \mid \lambda t.e \mid let_m\ t = e_1\ in\ e_2 \mid let_{pm}\ t = e_1\ in\ e_2 \mid let_p\ t = e_1\ in\ e_2$ where $c \in DC$, $f \in FS$ and t is a linear pattern. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ t_1 \ldots t_n$ where $c \in DC^n$, and *Higher order patterns* $HOPat = Pat \smallsetminus FOPat$. Expressions $h\ e_1 \ldots e_n$ are called *junk* if $h \in CS^k$ and $n > k$, and *active* if $h \in FS^k$ and $n \geq k$. $FV(e)$ is the set of variables in e which are not bound by any lambda or let expression and is defined in the usual way (notice that since our let expressions do not support recursive definitions the bindings of the pattern only affect e_2: $FV(let_*\ t = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \smallsetminus var(t))$. A *one-hole context* $\mathcal{C}$ is an expression with exactly one hole. A *data substitution* $\theta \in \mathcal{PS}ubst$ is a finite mapping from data variables to patterns: $[\overline{X_i/t_i}]$. Substitution application over data variables and expressions is defined in the usual way. A *program rule* is defined as $PRule \ni r ::= f\ t_1 \ldots t_n \to e\ (n \geq 0)$ where the set of patterns $\overline{t_i}$ is linear and $FV(e) \subseteq \bigcup_i var(t_i)$. Therefore, extra variables are not considered in this paper. A program is a set of program rules $Prog \ni \mathcal{P} ::= \{r_1; \ldots ; r_n\} (n \geq 0)$.

For the types we assume a denumerable set $\mathcal{TV}$ of *type variables* α and a countable alphabet $\mathcal{TC} = \bigcup_{n \in \mathbb{N}} \mathcal{TC}^n$ of *type constructors* C. The set of *simple types* is defined as $SType \ni \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid C\ \tau_1 \ldots \tau_n\ (C \in \mathcal{TC}^n)$. Based on simple types we define the set of *type-schemes* as $TScheme \ni \sigma ::= \tau \mid \forall \alpha.\sigma$. The set of *free type variables* (FTV) of a simple type τ is $var(\tau)$, and for type-schemes $FTV(\forall \overline{\alpha_i}.\tau) = FTV(\tau) \smallsetminus \{\overline{\alpha_i}\}$. A type-scheme $\forall \overline{\alpha_i}.\overline{\tau_n} \to \tau$ is *transparent* if $FTV(\overline{\tau_n}) \subseteq FTV(\tau)$. A *set of assumptions* $\mathcal{A}$ is $\{\overline{s_i : \sigma_i}\}$, where $s_i \in DC \cup FS \cup \mathcal{DV}$. Notice that the transparency of type-schemes for data constructors is not required in our setting, although that hypothesis is usually assumed in classical Damas & Milner type systems. If $(s_i : \sigma_i) \in \mathcal{A}$ we write $\mathcal{A}(s_i) = \sigma_i$. A *type substitution* $\pi \in \mathcal{TS}ubst$ is a finite mapping from type variables to simple types $[\overline{\alpha_i/\tau_i}]$. For sets of assumptions $FTV(\{\overline{s_i : \sigma_i}\}) = \bigcup_i FTV(\sigma_i)$. We will say a type-scheme σ is *closed* if $FTV(\sigma) = \emptyset$. Application of type substitutions to simple types is defined in the natural way, and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions in the obvious way. We will say σ is an *instance* of σ' if $\sigma = \sigma'\pi$ for some π. τ' is a *generic instance* of $\sigma \equiv \forall\overline{\alpha_i}.\tau$ if $\tau' = \tau[\overline{\alpha_i/\tau_i}]$ for some $\overline{\tau_i}$, and we write it $\sigma \succ \tau'$. We extend $\succ$ to a relation between type-schemes by saying that $\sigma \succ \sigma'$ iff every simple type such that is a generic instance of σ' is also a generic instance of σ. Then $\forall\overline{\alpha_i}.\tau \succ \forall\overline{\beta_i}.\tau[\overline{\alpha_i/\tau_i}]$ iff $\{\overline{\beta_i}\} \cap FTV(\forall\overline{\alpha_i}.\tau) = \emptyset$ [3]. Finally, τ' is a *variant* of $\sigma \equiv \forall\overline{\alpha_i}.\tau$ ($\sigma \succ_{var} \tau'$) if $\tau' = \tau[\overline{\alpha_i/\beta_i}]$ and $\overline{\beta_i}$ are fresh type variables.

3 Type Derivation

We propose a modification of Damas & Milner type system [4] with some differences. We have found convenient to separate the task of giving a regular Damas & Milner type and the task of checking critical variables. To do that we have defined two different type relations: $\vdash$ and $\vdash^\bullet$.

The basic typing relation $\vdash$ in the upper part of Fig. 2 is like the classical Damas & Milner's system but extended to handle the three different kinds of let expressions and the occurrence of patterns instead of variables in lambda and let expressions. We have also made the rules more syntax-directed so that the form of type derivations depends only on the form of the expression to be typed. $Gen(\tau, \mathcal{A})$ is the closure or generalization of τ wrt. $\mathcal{A}$ [4,3,19], which generalizes all the type variables of τ that do not appear free in $\mathcal{A}$. Formally: $Gen(\tau, \mathcal{A}) = \forall\overline{\alpha_i}.\tau$ *where* $\{\overline{\alpha_i}\} = FTV(\tau) \smallsetminus FTV(\mathcal{A})$. As can be seen, $[\text{LET}_m]$ and $[\text{LET}^h_{pm}]$ behave the same, and do not generalize any of the types τ_i for the variables X_i to give a type for the body. On the contrary, $[\text{LET}^X_{pm}]$ and $[\text{LET}_p]$ generalize the types given to the variables. Notice that if two variables share the same type in the set of assumptions $\mathcal{A}$, generalization will lose the connection between them. This fact can be seen with e_2 in Ex. 2. Although the type for both F and G can be $\alpha \rightarrow \alpha$ (with α a variable not appearing in $\mathcal{A}$) the generalization step will assign both the type-scheme $\forall\alpha.\alpha \rightarrow \alpha$, losing the connection between them. Fig. 3 shows a type derivation for the expression $\lambda(snd\ X).X$.

The $\vdash^\bullet$ relation (lower part of Fig. 2) uses $\vdash$ but enforces also the absence of critical variables. A variable X_i is *opaque* in t when it is possible to build a type derivation for t where the type assumed for X_i contains type variables which do not occur in the type derived for the pattern. The formal definition is as follows.

Definition 1 (Opaque variable of t wrt. $\mathcal{A}$). *Let t be a pattern that admits type wrt. a given set of assumptions $\mathcal{A}$. We say that $X_i \in \overline{X_i} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\exists\overline{\tau_i}, \tau$ s.t. $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau$ and $FTV(\tau_i) \nsubseteq FTV(\tau)$.*

Example 3 (Opaque variables of t wrt. $\mathcal{A}$).

– We will see that X is an opaque variable in $snd\ X$ wrt. any set of assumptions $\mathcal{A}_1$ containing the usual type-scheme for snd ($snd : \forall\alpha.\forall\beta.\alpha \rightarrow \beta \rightarrow \beta$) and

$$[\mathbf{ID}] \quad \frac{}{\mathcal{A} \vdash s : \tau} \quad \text{if } \begin{array}{l} s \in DC \cup FS \cup \mathcal{DV} \\ \wedge\ (s : \sigma) \in \mathcal{A} \wedge\ \sigma \succ \tau \end{array}$$

$$[\mathbf{APP}] \quad \frac{\begin{array}{c} \mathcal{A} \vdash e_1 : \tau_1 \to \tau \\ \mathcal{A} \vdash e_2 : \tau_1 \end{array}}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

$$[\Lambda] \quad \frac{\begin{array}{c} \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \\ \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau \end{array}}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau} \quad \text{if } \{\overline{X_i}\} = var(t)$$

$$[\mathbf{LET}_m] \quad \frac{\begin{array}{c} \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau_2 \end{array}}{\mathcal{A} \vdash \mathbf{let}_m\ t = e_1\ \mathbf{in}\ e_2 : \tau_2} \quad \text{if } \{\overline{X_i}\} = var(t)$$

$$[\mathbf{LET}^X_{pm}] \quad \frac{\begin{array}{c} \mathcal{A} \vdash e_1 : \tau_1 \\ \mathcal{A} \oplus \{X : Gen(\tau_1, \mathcal{A})\} \vdash e_2 : \tau_2 \end{array}}{\mathcal{A} \vdash \mathbf{let}_{pm}\ X = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$[\mathbf{LET}^h_{pm}] \quad \frac{\begin{array}{c} \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash h\ t_1 \ldots t_n : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau_2 \end{array}}{\mathcal{A} \vdash \mathbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \mathbf{in}\ e_2 : \tau_2} \quad \text{if } \begin{array}{l} \{\overline{X_i}\} = var(t_1 \ldots t_n) \\ \wedge\ h \in DC \cup FS \end{array}$$

$$[\mathbf{LET}_p] \quad \frac{\begin{array}{c} \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{\overline{X_i : Gen(\tau_i, \mathcal{A})}\} \vdash e_2 : \tau_2 \end{array}}{\mathcal{A} \vdash \mathbf{let}_p\ t = e_1\ \mathbf{in}\ e_2 : \tau_2} \quad \text{if } \{\overline{X_i}\} = var(t)$$

$$[\mathbf{P}] \quad \frac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash^{\bullet} e : \tau} \quad \text{if } critVar_{\mathcal{A}}(e) = \emptyset$$

Fig. 2. Rules of type system

Assuming $\mathcal{A} \equiv \{snd : \forall\alpha.\forall\beta.\alpha \to \beta \to \beta\}$ and $\mathcal{A}' \equiv \mathcal{A} \oplus \{X : \gamma\}$

$$[\Lambda]\ \frac{[\mathbf{APP}]\ \dfrac{(*)}{\mathcal{A} \oplus \{X : \gamma\} \vdash snd\ X : bool \to bool} \qquad [\mathbf{ID}]\ \dfrac{}{\mathcal{A}' \vdash X : \gamma}}{\mathcal{A} \vdash \lambda(snd\ X).X : (bool \to bool) \to \gamma}$$

where the type derivation for $(*)$ is:

$$[\mathbf{APP}]\ \frac{[\mathbf{ID}]\ \dfrac{}{\mathcal{A}' \vdash snd : \gamma \to bool \to bool} \qquad [\mathbf{ID}]\ \dfrac{}{\mathcal{A}' \vdash X : \gamma}}{\mathcal{A}' \vdash snd\ X : bool \to bool}$$

Fig. 3. Example of type derivation using $\vdash$

any type assumption for X. It is clear that *snd* X admits a type wrt. that $\mathcal{A}_1$, e.g. $bool \rightarrow bool$ (see Fig. 3). However we can build the type derivation $\mathcal{A}_1 \oplus \{X : \gamma\} \vdash snd\ X : bool \rightarrow bool$ such that $FTV(\gamma) = \{\gamma\} \nsubseteq \emptyset = FTV(bool \rightarrow bool)$.

– On the other hand we can see that X is not opaque in $snd\ [X, true]$. It corresponds to the intuition, since in this case the pattern itself fixes univocally the type of the variable X. Consider a set of assumptions $\mathcal{A}_2$ containing the usual type-schemes for *snd* and the list constructors, and the assumption $\{X : bool\}$. Clearly $snd\ [X, true]$ admits type wrt. $\mathcal{A}_2$. The only assumption for X that we can add to $\mathcal{A}_2$ in order to derive a type for $snd\ [X, true]$ is $\{X : bool\}$, otherwise the subpattern $[X, true]$ would not admit any type. Therefore any type derivation has to be of the shape $\mathcal{A}_2 \oplus \{X : bool\} \vdash snd\ [X, true] : \tau$, and obviously $FTV(bool) = \emptyset \subseteq FTV(\tau)$, for any τ.

Def. 1 is based on the existence of a certain type derivation, and therefore cannot be used as an effective check for the opacity of variables. Prop. 1 provides a more operational characterization of opacity that exploits the close relationship between $\vdash$ an type inference $\Vdash$ presented in Sect. 4.

Proposition 1. *$X_i \in \overline{X_i} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_g | \pi_g$ and $FTV(\alpha_i \pi_g) \nsubseteq FTV(\tau_g)$.*

We write $opaqueVar_{\mathcal{A}}(t)$ for set of opaque variables of t wrt. $\mathcal{A}$. Now, we can define the *critical variables* of an expression e wrt. $\mathcal{A}$ as those variables that, being opaque in a let or lambda pattern of e, are indeed used in e. Formally:

Definition 2 (Critical variables)

$critVar_{\mathcal{A}}(s) = \emptyset \quad if\ s \in DC \cup FS \cup \mathcal{DV}$

$critVar_{\mathcal{A}}(e_1\ e_2) = critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)$

$critVar_{\mathcal{A}}(\lambda t.e) = (opaqueVar_{\mathcal{A}}(t) \cap FV(e)) \cup critVar_{\mathcal{A}}(e)$

$critVar_{\mathcal{A}}(let_*\ t = e_1\ in\ e_2)$

$\qquad = (opaqueVar_{\mathcal{A}}(t) \cap FV(e_2)) \cup\ critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)$

Notice that the if we write the function *unpack* of Ex. 1 as $\lambda(snd\ X).X$, it is well-typed wrt. $\vdash$ using the usual type for *snd*. However it is ill-typed wrt. $\vdash^\bullet$ since X is a critical variable, i.e., it is an opaque variable in *snd* X and it occurs in the body of the λ-abstraction.

The typing relation $\vdash^\bullet$ has been defined in a modular way in the sense that the opacity check is kept separated from the regular Damas & Milner typing. Therefore it is easy to see that if every constructor and function symbol in program has a transparent assumption, then all the variables in patterns will be transparent, and so $\vdash^\bullet$ will be equivalent to $\vdash$. This happens in particular for those programs using only first order patterns and whose constructor symbols come from a Haskell (or Toy, Curry)-like `data` declaration.

3.1 Properties of the Typing Relations

The typing relations fulfill a set of useful properties. Here we use $\vdash^?$ for any of the two typing relations: $\vdash$ or $\vdash^\bullet$.

Theorem 1 (Properties of the typing relations)

a) *If* $\mathcal{A} \vdash^? e : \tau$ *then* $\mathcal{A}\pi \vdash^? e : \tau\pi$*, for any* $\pi \in \mathcal{TS}ubst$*.*
b) *Let* $s \in DC \cup FS \cup \mathcal{DV}$ *be a symbol not occurring in* e*. Then* $\mathcal{A} \vdash^? e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash^? e : \tau$*.*
c) *If* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e[X/e'] : \tau$*.*
d) *If* $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$*, then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$*.*

Part *a)* states that type derivations are closed under type substitutions. *b)* shows that type derivations for e depend only on the assumptions for the symbols in e. *c)* is a substitution lemma stating that in a type derivation we can replace a variable by an expression with the same type. Finally, *d)* establishes that from a valid type derivation we can change the assumption of a symbol for a more general type-scheme, and we still have a correct type derivation for the same type. Notice that this is not true wrt. the typing relation $\vdash^\bullet$ because a more general type can introduce opacity. For example the variable X is opaque in *snd* X with the usual type for *snd*, but with a more specific type such as $bool \rightarrow bool \rightarrow bool$ it is no longer opaque.

3.2 Subject Reduction

Subject reduction is a key property for type systems, meaning that evaluation does not change the type of an expression. This ensures that run-time type errors will not occur. Subject reduction is only guaranteed for *well-typed* programs, a notion that we formally define now.

Definition 3 (Well-typed program). *A program rule* $f\ t_1 \ldots t_n \rightarrow e$ *is well-typed wrt.* $\mathcal{A}$ *if* $\mathcal{A} \vdash^\bullet \lambda t_1 \ldots \lambda t_n.e : \tau$ *and* τ *is a variant of* $\mathcal{A}(f)$*. A program* $\mathcal{P}$ *is well-typed wrt.* $\mathcal{A}$ *if all its rules are well-typed wrt.* $\mathcal{A}$*. If* $\mathcal{P}$ *is well-typed wrt.* $\mathcal{A}$ *we write* $wt_{\mathcal{A}}(\mathcal{P})$*.*

Notice the use of the extended typing relation $\vdash^\bullet$ in the previous definition. This is essential, as we will explain later. Returning to Ex. 1, we can see that the program will not be well-typed because of the rule *unpack* (*snd* X) $\rightarrow X$, since $\lambda(snd\ X).X$ will be ill-typed wrt. the usual type for *snd*, as we explained before.

Although the restriction that the type of the lambda abstraction associated to a rule must be a variant of the type of the function symbol (and not an instance) might seem strange, it is necessary. Otherwise, the fact that a program is well-typed will not give us important information about the functions like the type of their arguments, and will make us to consider as well-typed undesirable programs like $\mathcal{P} \equiv \{f\ true \rightarrow true; f\ 2 \rightarrow false\}$ with the assumptions $\mathcal{A} \equiv \{f :: \forall\alpha.\alpha \rightarrow bool\}$. Besides, this restriction is implicitly considered in [6].

For subject reduction to be meaningful, a notion of evaluation is needed. In this paper we consider the *let-rewriting* relation of [11]. As can be seen, *let-rewriting* does not support let expressions with compound patterns. Instead of extending the semantics with this feature we propose a transformation from let-expressions with patterns to let-expressions with only variables (Fig. 4). There

$$\begin{array}{rl}
TRL(s) = s, & \text{if } s \in DC \cup FS \cup \mathcal{DV} \\
TRL(e_1\ e_2) = & TRL(e_1)\ TRL(e_2) \\
TRL(let_K\ X = e_1\ in\ e_2) = & let_K\ X = TRL(e_1)\ in\ TRL(e_2), \text{ with } K \in \{m, p\} \\
TRL(let_{pm}\ X = e_1\ in\ e_2) = & let_p\ X = TRL(e_1)\ in\ TRL(e_2) \\
TRL(let_m\ t = e_1\ in\ e_2) = & let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) \\
TRL(let_{pm}\ t = e_1\ in\ e_2) = & let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) \\
TRL(let_p\ t = e_1\ in\ e_2) = & let_p\ Y = TRL(e_1)\ in\ \overline{let_p\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2)
\end{array}$$

for $\{\overline{X_i}\} = var(t) \cap FV(e_2)$, $f_{X_i} \in FS^1$ fresh defined by the rule $f_{X_i}\ t \to X_i$, $Y \in \mathcal{DV}$ fresh, t a non variable pattern.

Fig. 4. Transformation rules of let expressions with patterns

are various ways to perform this transformation, which differ in the strictness of the pattern matching. We have chosen the alternative explained in [17] that does not demand the matching if no variable of the pattern is needed, but otherwise forces the matching of the whole pattern. This transformation has been enriched with the different kinds of let expressions in order to preserve the types, as is stated in Th. 2. Notice that the result of the transformation and the expressions accepted by *let-rewriting* only has let_m or let_p expressions, since without compound patterns let_{pm} is the same as let_p. Finally, we have added polymorphism annotations to let expressions (Fig. 5). Original **(Flat)** rule has been split into two, one for each kind of polymorphism. Although both behave the same from the point of view of values, the splitting is needed to guarantee type preservation. λ-abstractions have been omitted, since they are not supported by *let-rewriting*.

(Fapp) $f\ t_1\theta \dots t_n\theta \to^l r\theta$, if $(f\ t_1 \dots t_n \to r) \in \mathcal{P}$ and $\theta \in \mathcal{PS}ubst$

(LetIn) $e_1\ e_2 \to^l let_m\ X = e_2\ in\ e_1\ X$, if e_2 is an active expression, variable application, junk or *let* rooted expression, for X fresh.

(Bind) $let_K\ X = t\ in\ e \to^l e[X/t]$, if $t \in Pat$

(Elim) $let_K\ X = e_1\ in\ e_2 \to^l e_2$, if $X \notin FV(e_2)$

(Flat$_m$) $let_m\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3 \to^l let_K\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3)$, if $Y \notin FV(e_3)$

(Flat$_p$) $let_p\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3 \to^l let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(LetAp) $(let_K\ X = e_1\ in\ e_2)\ e_3 \to^l let_K\ X = e_1\ in\ e_2\ e_3$, if $X \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \to^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\ ]$, $e \to^l e'$ using any of the previous rules

where $K \in \{m, p\}$

Fig. 5. Higher order *let*-rewriting relation $\to^l$

Theorem 2 (Type preservation of the let transformation). *Assume* $\mathcal{A} \vdash^\bullet e : \tau$ *and let* $\mathcal{P} \equiv \{\overline{f_{X_i}\ t_i \to X_i}\}$ *be the rules of the projection functions needed in the transformation of* e *according to Fig. 4. Let also* $\mathcal{A}'$ *be the set of assumptions over that functions, defined as* $\mathcal{A}' \equiv \{\overline{f_{X_i} : Gen(\tau_{X_i}, \mathcal{A})}\}$*, where* $\mathcal{A} \Vdash^\bullet \lambda t_i.X_i : \tau_{X_i}|\pi_{X_i}$*. Then* $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet TRL(e) : \tau$ *and* $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$*.*

Th. 2 also states that the projection functions are well-typed. Then if we start from a well-typed program $\mathcal{P}$ wrt. $\mathcal{A}$ and apply the transformation to all its rules, the program extended with the projections rules will be well-typed wrt. the extended assumptions: $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P} \uplus \mathcal{P}')$. This result is straightforward, because $\mathcal{A}'$ does not contain any assumption for the symbols in $\mathcal{P}$, so $wt_{\mathcal{A}}(\mathcal{P})$ implies $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.

Th. 3 states the subject reduction property for a *let-rewriting* step, but its extension to any number of steps is trivial.

Theorem 3 (Subject Reduction). *If* $\mathcal{A} \vdash^\bullet e : \tau$ *and* $wt_{\mathcal{A}}(\mathcal{P})$ *and* $\mathcal{P} \vdash e \to^l e'$ *then* $\mathcal{A} \vdash^\bullet e' : \tau$*.*

For this result to hold it is essential that the definition of well-typed program relies on $\vdash^\bullet$. A counterexample can be found in Ex. 1, where the program would be well-typed wrt. $\vdash$ but the subject reduction property fails for *and* (*cast* 0) *true*.

The proof of the subject reduction property is based on the following lemma, an important auxiliary result about the instantiation of transparent variables. Intuitively it states that if we have a pattern t with type τ and we change its variables by other expressions, the only way to obtain the same type τ for the substituted pattern is by changing the transparent variables for expressions with the same type. This is not guaranteed with opaque variables, and that is why we forbid their use in expressions.

Lemma 1. *Assume* $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau$*, where* $var(t) \subseteq \{\overline{X_i}\}$*. If* $\mathcal{A} \vdash t[\overline{X_i/s_i}] : \tau$ *and* X_j *is a transparent variable of* t *wrt.* $\mathcal{A}$ *then* $\mathcal{A} \vdash s_j : \tau_j$*.*

4 Type Inference for Expressions

The typing relation $\vdash^\bullet$ lacks some properties that prevent its usage as a type-checker mechanism in a compiler for a functional logic language. First, in spite of the syntax-directed style, the rules for $\vdash$ and $\vdash^\bullet$ have a bad operational behavior: at some steps they need to guess a type. Second, the types related to an expression can be infinite due to polymorphism. Finally, the typing relation needs all the assumptions for the symbols in order to work. To overcome these problems, type systems usually are accompanied with a type inference algorithm which returns a valid type for an expression and also establishes the types for some symbols in the expression.

In this work we have given the type inference in Fig. 6 a relational style to show the similarities with the typing relation. But in essence, the inference rules represent an algorithm (similar to algorithm $\mathcal{W}$ [4,3]) which fails if any of the

[iID]
$$\frac{}{\mathcal{A} \Vdash s : \tau | id} \quad if \begin{array}{l} s \in DC \cup FS \cup \mathcal{DV} \\ \wedge\ (s : \sigma) \in \mathcal{A} \wedge \sigma \succ_{var} \tau \end{array}$$

[iAPP]
$$\frac{\begin{array}{c}\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \\ \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2\end{array}}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi | \pi_1\pi_2\pi} \quad if \begin{array}{l} \alpha\ fresh\ type\ variable \\ \wedge\ \pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha) \end{array}$$

[iΛ]
$$\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \\ (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t \Vdash e : \tau | \pi\end{array}}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau | \pi_t\pi} \quad if \begin{array}{l} \{\overline{X_i}\} = var(t) \\ \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \end{array}$$

[iLET$_m$]
$$\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \\ \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \\ (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2\end{array}}{\mathcal{A} \Vdash \mathbf{let}_m\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ \{\overline{X_i}\} = var(t) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \\ \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iLET$^X_{pm}$]
$$\frac{\begin{array}{c}\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \\ \mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2 | \pi_2\end{array}}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ X = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_1\pi_2}$$

[iLET$^h_{pm}$]
$$\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash h\ t_1 \ldots t_n : \tau_t | \pi_t \\ \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \\ (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2\end{array}}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ h \in DC \cup FS \wedge\ \{\overline{X_i}\} = var(h\ t_1 \ldots t_n) \\ \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iLET$_p$]
$$\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \\ \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \\ \mathcal{A}\pi_t\pi_1\pi \oplus \{\overline{X_i : Gen(\alpha_i\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\} \Vdash e_2 : \tau_2 | \pi_2\end{array}}{\mathcal{A} \Vdash \mathbf{let}_p\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ \{\overline{X_i}\} = var(t) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \\ \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iP]
$$\frac{\mathcal{A} \Vdash e : \tau | \pi}{\mathcal{A} \Vdash^{\bullet} e : \tau | \pi} \qquad if\ critVar_{\mathcal{A}\pi}(e) = \emptyset$$

Fig. 6. Inference rules

Assuming $\mathcal{A} \equiv \{snd : \forall\alpha.\forall\beta.\alpha \to \beta \to \beta\}$ and $\mathcal{A}' \equiv \mathcal{A} \oplus \{X : \gamma\}$

$$[i\Lambda]\ \dfrac{\textbf{[iAPP]}\ \dfrac{(*)}{\mathcal{A} \oplus \{X : \gamma\} \Vdash snd\ X : \epsilon \to \epsilon|\pi} \qquad \textbf{[iID]}\ \dfrac{}{\mathcal{A}' \Vdash X : \gamma|id}}{\mathcal{A} \Vdash \lambda(snd\ X).X : (\epsilon \to \epsilon) \to \gamma|\pi}$$

where the type inference for $(*)$ is:

$$\textbf{[iAPP]}\ \dfrac{\textbf{[iID]}\ \dfrac{}{\mathcal{A}' \Vdash snd : \delta \to \epsilon \to \epsilon|id} \qquad \textbf{[iID]}\ \dfrac{}{\mathcal{A}' \Vdash X : \gamma|id}}{\mathcal{A}' \Vdash snd\ X : \epsilon \to \epsilon|[\delta/\gamma, \zeta/\epsilon \to \epsilon] \equiv \pi}$$

where $\pi \equiv [\delta/\gamma, \zeta/\epsilon \to \epsilon]$ is the mgu of $\delta \to \epsilon \to \epsilon$ and $\gamma \to \zeta$
γ, δ, ϵ and ζ are fresh type variables

Fig. 7. Example of type inference using $\Vdash$

rules cannot be applied. This algorithm accepts a set of assumptions $\mathcal{A}$ and an expression e, and returns a simple type τ and a type substitution π. Intuitively, τ will be the "most general" type which can be given to e, and π the "minimum" substitution we have to apply to $\mathcal{A}$ in order to able to derive a type for e. Fig. 7 contains an example of type inference for the expression $\lambda(snd\ X).X$.

Th. 4 shows that the type and substitution found by the inference are correct, i.e., we can build a type derivation for the same type if we apply the substitution to the assumptions.

Theorem 4 (Soundness of $\Vdash^?$). $\mathcal{A} \Vdash^? e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash^? e : \tau$

Th. 5 expresses the completeness of the inference process. If we can derive a type for an expression applying a substitution to the assumptions, then inference will succeed and will find a type and a substitution which are the most general ones.

Theorem 5 (Completeness of $\Vdash$ wrt $\vdash$). *If $\mathcal{A}\pi' \vdash e : \tau'$ then $\exists\tau, \pi, \pi''$. $\mathcal{A} \Vdash e : \tau|\pi \wedge \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \wedge \tau\pi'' = \tau'$.*

A result similar to Th. 5 cannot be obtained for $\Vdash^\bullet$ because of critical variables, as the following example 4 shows.

Example 4 (Inexistence of a most general typing substitution). Let $\mathcal{A} \equiv \{snd' : \alpha \to bool \to bool\}$ and consider the following two valid derivations $\mathcal{D}_1 \equiv \mathcal{A}[\alpha/bool] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to bool$ and $\mathcal{D}_2 \equiv \mathcal{A}[\alpha/int] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to int$. It is clear that there is not a substitution more general than $[\alpha/bool]$ and $[\alpha/int]$ which makes possible a type derivation for $\lambda(snd'\ X).X$. The only substitution more general than these two will be $[\alpha/\beta]$ (for some β), converting X in a critical variable.

In spite of this, we will see that $\Vdash^\bullet$ is still able to find the most general substitution when it exists. To formalize that, we will use the notion of $\Pi^\bullet_{\mathcal{A},e}$, which denotes the set collecting all type substitution π such that $\mathcal{A}\pi$ gives some type to e.

Definition 4 (Typing substitutions of e)
$\Pi^{\bullet}_{\mathcal{A},e} = \{\pi \in \mathcal{TS}ubst \mid \exists\tau \in SType.\ \mathcal{A}\pi \vdash^{\bullet} e : \tau\}$

Now we are ready to formulate our result regarding the maximality of $\Vdash^{\bullet}$.

Theorem 6 (Maximality of $\Vdash^{\bullet}$)

a) $\Pi^{\bullet}_{\mathcal{A},e}$ *has a maximum element* $\iff \exists\tau_g \in SType, \pi_g \in \mathcal{TS}ubst.\ \mathcal{A} \Vdash^{\bullet} e : \tau_g|\pi_g$.
b) *If* $\mathcal{A}\pi' \vdash^{\bullet} e : \tau'$ *and* $\mathcal{A} \Vdash^{\bullet} e : \tau|\pi$ *then exists a type substitution* π'' *such that* $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ *and* $\tau' = \tau\pi''$.

5 Type Inference for Programs

In the functional programming setting, type inference does not need to distinguish between programs and expressions, because the program can be incorporated in the expression by means of let expressions and λ-abstractions. This way, the results given for expressions are also valid for programs. But in our framework it is different, because our semantics (*let-rewriting*) does not support λ-abstractions and our let expressions do not define new functions but only perform pattern matching. Thereby in our case we need to provide an explicit method for inferring the types of a whole program. By doing so, we will also provide a specification closer to implementation.

The type inference procedure for a program takes a set of assumptions $\mathcal{A}$ and a program $\mathcal{P}$ and returns a type substitution π. The set $\mathcal{A}$ must contain assumptions for all the symbols in the program, even for the functions defined in $\mathcal{P}$. We want to reflect the fact that in practice some defined functions may come with an explicit type declaration. Indeed this is a frequent way of documenting a program. Furthermore, type declarations are sometimes a real need, for instance if we want the language to support *polymorphic recursion* [16,10]. Therefore, for some of the functions –those for which we want to infer types– the assumption will be simply a fresh type variable, to be instantiated by the inference process. For the rest, the assumption will be a closed type-scheme, to be checked by the procedure.

Definition 5 (Type Inference of a Program). *The procedure* $\mathcal{B}$ *for type inference of a program* $\{rule_1, \ldots, rule_m\}$ *is defined as:*

$$\mathcal{B}(\mathcal{A}, \{rule_1, \ldots, rule_m\}) = \pi, \textit{if}$$

1. $\mathcal{A} \Vdash^{\bullet} (\varphi(rule_1), \ldots, \varphi(rule_m)) : (\tau_1, \ldots, \tau_m)|\pi$.
2. *Let* $f^1 \ldots f^k$ *be the function symbols of the rules* $rule_i$ *in* $\mathcal{P}$ *such that* $\mathcal{A}(f^i)$ *is a closed type-scheme, and* τ^i *the type obtained for* $rule_i$ *in step 1. Then* τ^i *must be a variant of* $\mathcal{A}(f^i)$.

φ is a transformation from rules to expressions defined as:

$$\varphi(f\ t_1 \ldots t_n \to e) = pair\ \lambda t_1. \ldots \lambda t_n.e\ f$$

where **()** *is the usual tuple constructor, with type* **()** $: \forall\overline{\alpha_i}.\alpha_1 \to \ldots \alpha_m \to (\alpha_1, \ldots, \alpha_m)$*; and* ***pair*** *is a special constructor of tuples of two elements of the same type, with type* ***pair*** $: \forall\alpha.\alpha \to \alpha \to \alpha$*.*

Example 5 (Type Inference of Programs).

– Consider the program $\mathcal{P}$ consisting in the rules $\{ugly\ true \to true, ugly\ 0 \to true\}$ and the set of assumptions $\mathcal{A} \equiv \{ugly : \forall\alpha.\alpha \to bool\}$. Our intuition advises us to reject this program because the type of *ugly* expresses parametric polymorphism, and the rules are not parametric but defined for arguments whose types are not compatible. Using procedure $\mathcal{B}$ we will first infer the type for the expression associated to the program, getting

  $\mathcal{A} \Vdash^{\bullet} (pair\ \lambda true.true\ ugly, pair\ \lambda 0.true\ ugly) : (bool \to bool, int \to bool)|\pi$

  for some π that affects only type variables generated during the inference. Since *ugly* has a closed type-scheme in $\mathcal{A}$ then we will check that the types $bool \to bool$ and $int \to bool$ inferred for its rules are variants of $\forall\alpha.\alpha \to bool$. This check will fail, therefore the procedure $\mathcal{B}$ will reject the program.

– Consider the program $\mathcal{P} \equiv \{and\ true\ X \to X, and\ false\ X \to false, id\ X \to X\}$ and the set of assumptions $\mathcal{A} \equiv \{and : \beta, id : \forall\alpha.\alpha \to \alpha\}$. In this case we want to infer the type for *and* (instantiating type variable β) and check that the type for *id* is correct. Using procedure $\mathcal{B}$, in the first step we infer the type for the expression associated to the program:

  $\mathcal{A} \Vdash^{\bullet} (pair\ \lambda true.\lambda X.X\ and, pair\ \lambda false.\lambda X.false\ and, pair\ \lambda X.X\ id) : (bool \to bool \to bool, bool \to bool \to bool, \gamma \to \gamma) : [\beta/bool \to bool \to bool]$[1]

  Therefore the type inferred for *and* would be the expected one: $bool \to bool \to bool$. Since *id* has a closed type-scheme in $\mathcal{A}$ then the second step will check the type inferred $\gamma \to \gamma$ is a variant of $\forall\alpha.\alpha \to \alpha$. The check is correct, therefore $\mathcal{B}$ succeeds with the substitution $[\beta/bool \to bool \to bool]$.

The procedure $\mathcal{B}$ has two important properties. It is sound: if the procedure $\mathcal{B}$ finds a substitution π then the program $\mathcal{P}$ is well-typed with respect to the assumptions $\mathcal{A}\pi$ (Th. 7). And second, if the procedure $\mathcal{B}$ succeeds it finds the most general typing substitution (Th. 8). It is not true in general that the existence of a well-typing substitution π' implies the existence of a most general one. A counterexample of this fact is very similar to Ex. 4.

Theorem 7 (Soundness of $\mathcal{B}$). *If $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then $wt_{\mathcal{A}\pi}(\mathcal{P})$.*

Theorem 8 (Maximality of $\mathcal{B}$). *If $wt_{\mathcal{A}\pi'}(\mathcal{P})$ and $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then $\exists\pi''$ such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.*

[1] Note that the bindings for type variables which are not free in $\mathcal{A}$ have been omitted here for the sake of conciseness.

Notice that types inferred for the functions are simple types. In order to obtain type-schemes we need and extra step of generalization, as discussed in the next section.

5.1 Stratified Type Inference of a Program

It is known that splitting a program into blocks of mutually recursive functions and inferring the types in order may reduce the need of providing explicit type-schemes. This situation is shown in the next example.

Example 6 (Program Inference vs Stratified Inference).

$\mathcal{A} \equiv \{true : bool, 0 : int, id : \alpha, f : \beta, g : \gamma\}$
$\mathcal{P} \equiv \{id\ X \to X; f \to id\ true; g \to id\ 0\}$
$\mathcal{P}_1 \equiv \{id\ X \to X\}$, $\mathcal{P}_2 \equiv \{f \to id\ true\}$, $\mathcal{P}_3 \equiv \{g \to id\ 0\}$

An attempt to apply the procedure $\mathcal{B}$ to infer types for the whole program fails because it is not possible for id to have types $bool \to bool$ and $int \to int$ at the same time. We will need to provide explicitly the type-scheme for $id : \forall\alpha.\alpha \to \alpha$ in order to the type inference to succeed, yielding types $f : bool \to bool$ and $g : int \to int$. But this is not necessary if we first infer types for $\mathcal{P}_1$, obtaining $\delta \to \delta$ for id which will be generalized to $\forall\delta.\delta \to \delta$. With this assumption the type inference for both programs $\mathcal{P}_2$ and $\mathcal{P}_3$ will succeed with the expected types.

A general *stratified inference* procedure can be defined in terms of the basic inference $\mathcal{B}$. First, it calculates the graph of strongly connected components from the dependency graph of the program, using e.g. Kosaraju or Tarjan's algorithm [20]. Each strongly connected component will contain mutually dependent functions. Then it will infer types for every component (using $\mathcal{B}$) in topological order, generalizing the obtained types before following with the next component.

Although stratified inference needs less explicit type-schemes, programs involving polymorphic recursion still require explicit type-schemes in order to infer their types.

6 Conclusions and Future Work

In this paper we have proposed a type system for functional logic languages based on Damas & Milner type system. As far as we know, prior to our work only [6] treats with technical detail a type system for functional logic programming. Our paper makes clear contributions when compared to [6]:

– By introducing the notion critical variables, we are more liberal in the treatment of opaque variables, but still preserving the essential property of subject reduction; moreover, this liberality extends also to data constructors, dropping the traditional restriction of transparency required to them. This is somehow similar to what happens with *existential types* [14] or *generalized abstract datatypes* [9], a connection that we plan to further investigate in the future.

- Our type system considers local pattern bindings and λ-abstractions (also with patterns), that were missing in [6]. In addition to that, we have made a rather exhaustive analysis and formalization of different possibilities for polymorphism in local bindings.
- Subject reduction was proved in [6] wrt. a narrowing calculus. Here we do it wrt. an small-step operational semantics closer to real computations.
- In [6] programs came with explicit type declarations. Here we provide algorithms for inferring types for programs without such declarations that can became part of the type stage of a FL compiler.

We have in mind several lines for future work. As an immediate task we plan to implement and integrate the stratified type inference into the $\mathcal{TOY}$ [12] compiler. Apart from the relation to existential types mentioned above, we are interested in other known extensions of type system, like type classes or generic programming. We also want to generalize the subject reduction property to narrowing, using *let narrowing* reductions of [11], and taking into account known problems [6,1] in the interaction of HO narrowing and types. Handling extra variables (variables occurring only in right hand sides of rules) is another challenge from the viewpoint of types.

References

1. Antoy, S., Tolmach, A.P.: Typed higher-order narrowing without higher-order strategies. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 335–353. Springer, Heidelberg (1999)
2. Brassel, B.: Two to three ways to write an unsafe type cast without importing unsafe - Post to the Curry mailing list (May 2008), http://www.informatik.uni-kiel.de/~curry/listarchive/0705.html
3. Damas, L.: Type Assignment in Programming Languages. PhD thesis, University of Edinburgh (April 1985)
4. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proc. Symposium on Principles of Programming Languages (POPL 1982), pp. 207–212 (1982)
5. González-Moreno, J., Hortalá-González, T., Rodríguez-Artalejo, M.: A higher order rewriting logic for functional logic programming. In: Proc. International Conference on Logic Programming (ICLP 1997), pp. 153–167. MIT Press, Cambridge (1997)
6. González-Moreno, J., Hortalá-González, T., Rodríguez-Artalejo, M.: Polymorphic types in functional logic programming. Journal of Functional and Logic Programming 2001/S01, 1–71 (2001)
7. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
8. Hanus, M. (ed.): Curry: An integrated functional logic language (version 0.8.2) (March 2006), http://www.informatik.uni-kiel.de/~curry/report.html
9. Jones, S.L.P., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. In: Proc. 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, pp. 50–61. ACM, New York (2006)
10. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: Type reconstruction in the presence of polymorphic recursion. ACM Trans. Program. Lang. Syst. 15(2), 290–311 (1993)

11. López-Fraguas, F.J., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Rewriting and call-time choice: the HO case. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 147–162. Springer, Heidelberg (2008)
12. López-Fraguas, F., Sánchez-Hernández, J.: $\mathcal{TOY}$: A multiparadigm declarative system. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
13. Martin-Martin, E.: Advances in type systems for functional-logic programming. Master's thesis, Universidad Complutense de Madrid (July 2009), `http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf`
14. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Trans. Program. Lang. Syst. 10(3), 470–502 (1988)
15. Moreno-Navarro, J.J., Mariño, J., del Pozo-Pietro, A., Herranz-Nieva, Á., García-Martín, J.: Adding type classes to functional-logic languages. In: 1996 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE 1996, pp. 427–438 (1996)
16. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 217–228. Springer, Heidelberg (1984)
17. Peyton Jones, S.: The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs (1987)
18. Pierce, B.P.: Advanced topics in types and programming languages. MIT Press, Cambridge (2005)
19. Reade, C.: Elements of Functional Programming. Addison-Wesley, Reading (1989)
20. Sedgewick, R.: Algorithms in C++, Part 5: Graph Algorithms, pp. 205–216. Addison-Wesley Professional, Reading (2002)

A Simple Region Inference Algorithm for a First-Order Functional Language*

Manuel Montenegro, Ricardo Peña, and Clara Segura

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
montenegro@fdi.ucm.es, {ricardo,csegura}@sip.ucm.es

Abstract. *Safe* is a first-order eager language with facilities for programmer controlled destruction and copying of data structures. It provides also *regions*, i.e. disjoint parts of the heap, where the program allocates data structures. The runtime system does not need a garbage collector and all allocation/deallocation actions are done in constant time. The language is aimed at inferring and certifying upper bounds for memory consumption in a Proof Carrying Code environment. Some of its analyses have been presented elsewhere [7,8]. In this paper we present an inference algorithm for annotating programs with regions which is both simpler to understand and more efficient than other related algorithms. Programmers are assumed to write programs and to declare datatypes without any reference to regions. The algorithm decides the regions needed by every function. It also allows polymorphic recursion with respect to regions. We show convincing examples of programs before and after region annotation, prove the correctness and optimality of the algorithm, and give its asymptotic cost.

1 Introduction

Safe[1] [7] was introduced as a research platform for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements. The final aim is to be able to infer —at compile time— safe upper bounds on memory consumption for most *Safe* programs. The compiler produces Java bytecode as a target language, so that *Safe* programs can be executed in most mobile devices and web browsers.

In most functional languages memory management is delegated to the runtime system. Fresh heap memory is allocated during program evaluation as long as there is enough free memory available. Garbage collection interrupts program execution in order to copy or mark the live part of the heap so that the rest is considered as free. This does not avoid memory exhaustion if not enough free memory is recovered to continue execution. In that case the program simply

* Work supported by the Ministry of Science grants AP2006-02154, TIN2008-06622-C03-01/TIN (STAMP), and the Madrid Government grant S-0505/TIC/0407 (PROMESAS).

[1] http://dalila.sip.ucm.es/safe

aborts. The main advantage of this approach is that programmers do not have to bother about low level details concerning memory management. Its main disadvantages are:

1. The time delay introduced by garbage collection may prevent the program from providing an answer in a required reaction time.
2. Memory exhaustion may provoke unacceptable personal or economic damage to program users.
3. The programmer cannot easily reason about memory consumption.

These reasons make garbage collectors not very convenient for programming small devices. A possibility is to use heap *regions*, which are disjoint parts of the heap that are dynamically allocated and deallocated. Much work has been done in order to incorporate regions in functional languages. They were introduced by Tofte and Talpin [13,14] in MLKit by means of a nested **letregion** construct inferred by the compiler. The drawbacks of nested regions are well-known and they have been discussed in many papers (see e.g. [4]). The main problem is that in practice data structures do not always have the nested lifetimes required by the stack-based region discipline.

In order to overcome this limitation several mechanisms have been proposed. An extension of Tofte and Talpin's work [2,11] allows to *reset* all the data structures in a region, without deallocating the whole region. The AFL system [1] inserts (as a result of an analysis) allocation and deallocation commands separated from the **letregion** construct, which now only brings new regions into scope. In both cases, a deep knowledge about the hidden mechanism is needed in order to optimize the memory usage. In particular, it is required to write copy functions in the program which are difficult to justify without knowing the annotations inferred later by the compiler.

Another more explicit approach is to introduce a language construct to free heap memory. Hofmann and Jost [5] introduce a pattern matching construct which destroys individual constructor cells than can be reused by the memory management system. This allows the programmer to control the memory consumed by the program and to reason about it. However, this approach gives the programmer the whole responsibility for reusing memory, unless garbage collection is used.

In order to overcome the problems related to nested regions, our functional language *Safe* has a semi-explicit approach to memory control: it combines implicit regions with explicit destructive pattern matching, which deallocates individual cells of a data structure. This feature avoid the use of explicit copy functions of other systems. In *Safe*, regions are allocated/deallocated by following a stack discipline associated with function calls and returns. Each function call allocates a local working region, which is deallocated when the function returns. Region management does not add a significant runtime overhead because all its related operations run in constant time (see Sec. 2.3).

Notice that regions and explicit destruction are orthogonal mechanisms: we could have destruction without regions and the other way around. This combination of explicit destruction and implicit regions is novel in the functional programming field. However, destructive pattern matching is not relevant to this paper. More details about it can be found in [7,8]

Due to the aim of inferring memory consumption upper bounds, at this moment *Safe* is first-order. Its syntax is a (first-order) subset of Haskell extended with destructive pattern matching. Due to this limitation, region inference can be expected to be simpler and more efficient than that of MLKit. Their algorithm runs in time $O(n^4)$ in the worst case, where n is the size of the term, including in it the Hindley-Milner type annotations. The explanation of the algorithm and of its correctness arguments [10] needed around 40 pages of dense writing. So, it is not an easy task to incorporate the MLKit ideas into a new language.

The contribution of this paper is a simple region inference algorithm for *Safe*. It allows polymorphic recursion w.r.t. regions (region-polymorphic recursion, in the following). Hindley-Milner type inference is in general undecidable under polymorphic recursion, but when restricting to region-polymorphic recursion it becomes decidable. Our algorithm runs in $O(n)$ time in the worst case (being n as above) if region-polymorphic recursion is not inferred. If the latter appears, the algorithm needs $O(n^2)$ time in the worst case. Moreover, the first phase of the algorithm can be directly integrated in the usual Hindley-Milner type inference algorithm, just by considering regions as ordinary polymorphic type variables. The second phase involves very simple set operations and the computation of a fixpoint. Unlike [10], termination is always guaranteed without special provisions. There, they had to sacrifice principal types in order to ensure termination. Due to its simplicity, we believe that the algorithm can be easily reused in a different first-order functional language featuring Hindley-Milner types.

The plan of the paper is as follows: In Sec. 2 we summarize the language concepts and part of its big-step operational semantics. In Sec. 3 the region inference algorithm is presented in detail, including its correctness and cost. Section 4 shows some examples of region inference with region polymorphic recursion. Finally, Sec. 5 compares this work with other functional languages with memory management facilities.

2 Language Concepts and Inference Examples

2.1 Operational Semantics

In Fig. 1 we show a simplified version of the *Safe* language without the destruction facilities but with explicit region arguments and region types. A program is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e, using them, whose value is the program result. The abbreviation $\overline{x_i}^n$ stands for $x_1 \cdots x_n$. We use $a, a_i, \ldots$ to denote atoms, i.e. either program variables or basic constants. The former are denoted by $x, x_i, \ldots$ and the latter by $c, c_i \ldots$ etc. Region arguments $r, r_i \ldots$ occur in function definitions and in function and constructor applications. They are containers used at runtime to pass region values around. Region values k are runtime numbers denoting actual regions in the region stack, and region types ρ are static annotations assigned to region variables and occuring in type declarations.

$$
\begin{array}{lll}
\mathit{prog} \rightarrow \overline{\mathit{data}_i}^{n}; \overline{\mathit{dec}_j}^{m}; e & & \\
\mathit{data} \rightarrow \mathbf{data}\ T\ \overline{\alpha_i}^{n}\ @\ \overline{\rho_j}^{m} = \overline{C_k\ \overline{t_{ks}}^{n_k}\ @\ \rho_m}^{l} & & \{\text{recursive, polymorphic data type}\} \\
\mathit{dec} \ \ \rightarrow & f\ \overline{x_i}^{n}\ @\ \overline{r_j}^{l} = e & \{\text{recursive, polymorphic function}\} \\
e \ \ \ \ \rightarrow & a & \{\text{atom: literal } c \text{ or variable } x\} \\
 & \mid f\ \overline{a_i}^{n}\ @\ \overline{r_j}^{l} & \{\text{function application}\} \\
 & \mid C\ \overline{a_i}^{n}\ @\ r & \{\text{constructor application}\} \\
 & \mid \ldots & \text{let, case} \ldots
\end{array}
$$

Fig. 1. Simplified *Safe*

Safe was designed in such a way that the compiler has a complete control on where and when memory allocation and deallocation actions will take place at runtime. The smallest memory unit is the **cell**, a contiguous memory space big enough to hold any data construction. A cell contains the mark of the constructor and a representation of the free variables to which the constructor is applied. These may consist either of basic values or of pointers to other constructions. It is allocated at constructor application time and can be deallocated by destructive pattern matching. A **region** is a collection of cells, not necessarily contiguous in memory. Regions are allocated/deallocated by following a stack discipline associated with function calls and returns. Each function call allocates a local working region, which is deallocated when the function returns.

In Fig. 2 we show those rules of the big-step operational semantics which are relevant with respect to regions. We use $v, v_i, \ldots$ to denote values, i.e. either heap pointers or basic constants, and $p, p_i, q, \ldots$ to denote heap pointers.

A judgement of the form $E \vdash h, k, e \Downarrow h', k, v$ means that expression e is successfully reduced to normal form v under runtime environment E and heap h with $k+1$ regions, ranging from 0 to k, and that a final heap h' with $k+1$ regions is produced as a side effect. Runtime environments E map program variables to values and region variables to actual region identifiers. We adopt the convention that for all E, if c is a constant, $E(c) = c$.

A heap h is a finite mapping from fresh variables p to construction cells w of the form $(j, C\,\overline{v_i}^{n})$, meaning that the cell resides in region j. Actual region identifiers j are just natural numbers denoting the offset of the region from the bottom of the region stack. Formal regions appearing in a function body are either region variables r corresponding to formal arguments or the constant *self*, which represents the local working region. By $h \uplus [p \mapsto w]$ we denote the disjoint

$$
\frac{(f\ \overline{x_i}^{n} @\ \overline{r_j}^{m} = e) \in \Sigma \qquad [\overline{x_i \mapsto E(a_i)}^{n}, \overline{r_j \mapsto E(r'_j)}^{m}, \mathit{self} \mapsto k+1] \vdash h, k+1, e \Downarrow h', k+1, v}{E \vdash h, k, f\ \overline{a_i}^{n} @\ \overline{r'_j}^{m} \Downarrow h' \mid_k, k, v}\ [\mathit{App}]
$$

$$
\frac{j \leq k \qquad \mathit{fresh}(p)}{E[r \mapsto j, \overline{a_i \mapsto v_i}^{n}] \vdash h, k, C\,\overline{a_i}^{n} @ r \Downarrow h \uplus [p \mapsto (j, C\,\overline{v_i}^{n})], k, p}\ [\mathit{Cons}]
$$

Fig. 2. Operational semantics of *Safe* expressions

union of heap h with the binding $[p \mapsto w]$. By $h\mid_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k.

The semantics of a program is the semantics of the main expression e in an environment Σ, which is the set containing all the function and data declarations.

Rule *App* shows when a new region is allocated. Notice that the body of the function is executed in a heap with $k + 2$ regions. The formal identifier *self* is bound to the newly created region $k + 1$ so that the function body may create data structures in this region or pass this region as a parameter to other function calls. Before returning from the function, all cells created in region $k+1$ are deleted. In rule *Cons* a fresh construction cell is allocated in the heap.

2.2 Region Annotations

The aim of the region inference algorithm is to annotate both the program and the types of the functions with region variables and region type variables respectively. Before explaining the inference algorithm we show some illustrative examples.

Regions are essentially the parts of the heap where the data structures live. We will consider as a **data structure** (DS) the set of cells obtained by starting at one cell considered as the root, and taking the transitive closure of the relation $C_1 \rightarrow C_2$, where C_1 and C_2 are cells of the same type T, and in C_1 there is a pointer to C_2. That means that, for instance in a list of type `[[a]]`, we consider as a DS all the cells belonging to the outermost list, but not those belonging to the individual innermost lists. Each one of the latter constitute a DS living in a possibly different region from the outermost's one. However, since all the innermost lists have the same type, they will be forced to reside in the same region. A DS completely resides in one region. A DS can be part of another DS, or two DSs can share a third one. The basic values —integers, booleans, etc.— do not allocate cells in regions. They live inside the cells of DSs, or in the stack.

These decisions are reflected in the way the type system deals with datatype definitions. Polymorphic algebraic data types are defined through **data** declarations as the following one:

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

The types assigned by the compiler to constructors include an additional argument indicating the region where the constructed values of that type are allocated. In the example, the compiler infers:

$$\textbf{data}\ \mathit{Tree}\ a\ @\ \rho = \mathit{Empty}@\ \rho \mid \mathit{Node}\ (\mathit{Tree}\ a\ @\ \rho)\ a\ (\mathit{Tree}\ a\ @\ \rho)@\ \rho$$

where ρ is the type of the region argument given to the constructors. After region inference, constructions appear in the annotated text with an additional argument `r` that will be bound at runtime to an actual region, as in `Node lt x rt @ r`. Constructors are polymorphic in region arguments, meaning that they can be applied to any actual region. But, due to the above type restrictions, and in the case of `Node`, this region must be the same where both the left tree `lt` and the right tree `rt` live.

Several regions can be inferred when nested types are used, as different components of the data structure may live in different regions. For instance, in the declaration

```
data Table a b = TBL [(a,b)]
```

the following three region types will be inferred for the `Table` datatype:

$$\textbf{data}\ \textit{Table}\ a\ b\ @\ \rho_1\ \rho_2\ \rho_3 = \textit{TBL}\ ([(a,b)@\ \rho_1]@\ \rho_2)@\ \rho_3$$

In that case we adopt the convention that the last region type in the list is the outermost one where the constructed values of the datatype are to be allocated.

After region inference, function applications are annotated with the additional region arguments which the function uses to construct DSs. For instance, in the definition

```
concat []     ys  = ys
concat (x:xs) ys  = x : concat xs ys
```

the compiler infers the type *concat* $:: \forall a \rho_1 \rho_2 . [a]@\rho_1 \to [a]@\rho_2 \to \rho_2 \to [a]@\rho_2$ and annotates the text as follows:

```
concat []     ys @ r = ys
concat (x:xs) ys @ r = (x : concat xs ys @ r) @ r
```

The region of the output list and that of the second input list must be the same due to the sharing between both lists introduced by the first equation. Functions are also polymorphic in region types, i.e. they can accept as arguments any actual regions provided that they satisfy the type restrictions (for instance, in the case of `concat`, that the second and the output lists must live in the same region). Sometimes, several region arguments are needed as in:

```
partition y [] = ([],[])
partition y (x:xs) | x <= y = (x:ls,gs)
                   | x >  y = (ls  ,x:gs)
                 where (ls,gs) = partition y xs
```

The inferred type is *partition* $:: \forall \rho_1 \rho_2 \rho_3 \rho_4 . \textit{Int} \to [\textit{Int}]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([\textit{Int}]@\rho_2, [\textit{Int}]@\rho_3)@\rho_4$. The algorithm splits the output in as many regions as possible. This gives more general types and allows the garbage to be deallocated sooner.

When a function body is executing, the *live* regions are the working regions of all the active function calls leading to this one. The live regions in scope are those where the argument DSs live (for reading), those received as additional arguments (for reading and writing) and the own *self* region. The following example builds an intermediate tree not needed in the output:

```
treesort xs = inorder (makeTree xs)
```

where the inferred types are as follows:

$$\begin{array}{ll} \mathit{makeTree} & :: \forall a \rho_1 \rho_2 . [a]@\rho_1 \rightarrow \rho_2 \rightarrow \mathit{Tree}\ a@\rho_2 \\ \mathit{inorder} & :: \forall a \rho_1 \rho_2 . \mathit{Tree}\ a@\rho_1 \rightarrow \rho_2 \rightarrow [a]@\rho_2 \\ \mathit{treesort} & :: \forall a \rho_1 \rho_2 . [a]@\rho_1 \rightarrow \rho_2 \rightarrow [a]@\rho_2 \end{array}$$

After region inference, the definition is annotated as follows:

```
treesort xs @ r = inorder (makeTree xs @ self) @ r
```

i.e. the intermediate tree is created in the *self* region and it is deallocated upon termination of `treesort`.

The region inference mechanism will not lead to rejecting programs. It always succeeds although, of course, it will not be able to detect all garbage. Section 3 explains how the algorithm works and shows that it is optimal in the sense that it assigns as many DS as possible to the *self* region of the function at hand.

2.3 Region Implementation

As we said above, the heap is implemented as a stack of regions. Each region is pushed initially empty, this action being associated with a *Safe* function invocation. During function execution new cells can be added to, or removed from, any active region as a consequence of constructor applications and destructive pattern matching. Upon function termination the whole topmost region is deallocated. In Fig. 3 we show the main interface between a running *Safe* program and the Memory Management System (MMS). It is written in Java since the code generated by the *Safe* compiler is Java bytecode. The MMS maintains a pool of fresh cells, so that 'allocating' and 'deallocating' a cell respectively mean removing it from, or adding it to the pool.

Notice that access to an arbitrary region is needed in *InsertCell*, whereas *ReleaseCell* is only provided with the cell pointer as an argument. We have implemented all the methods running in constant time by representing the regions and the pool as circular doubly-chained lists. Removing a region amounts to joining two circular lists, which can obviously be done in constant time. The region stack is represented by a static array of dynamic lists, so that constant time access to each region is provided. Fig. 4 shows a picture of the heap.

Tail recursive functions can very easily be detected at compile time so that a special translation for them would not push a new empty region at each invocation, but instead reuse the current topmost region. This translation (not yet implemented in our compiler) would not avoid consuming new cells at each invocation

void PushRegion ()	-- creates a top empty region
void PopRegion ()	-- removes the topmost region
cell ReserveCell ()	-- returns a fresh cell
void InsertCell (p, j)	-- inserts cell p into region j
void ReleaseCell (p)	-- releases cell p

Fig. 3. The interface of the *Safe* Memory Management System

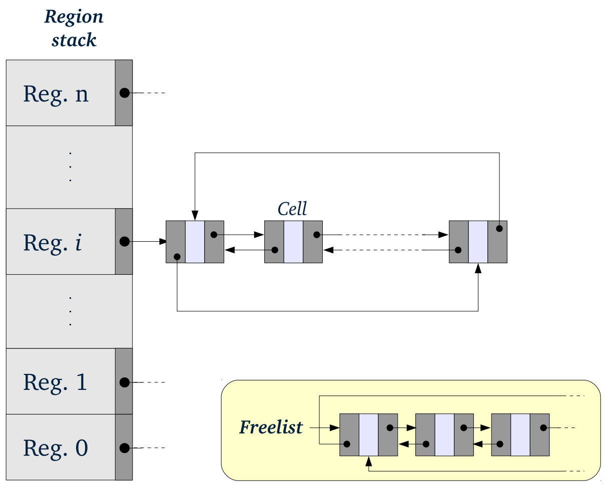


Fig. 4. A picture of the Safe Virtual Machine heap and fresh cells pool

but at least would consume a constant stack space in the region stack. Consuming constant heap space in tail recursive functions is not feasible in general because any function invocation may freely access regions below the topmost one.

The *Safe* virtual machine has also a conventional stack where local variables are kept. The code generated for function invocation guarantees that tail recursive functions always consume constant stack space. In this respect, no special translation is needed.

3 The Region Inference Algorithm

The main correctness requirement to the region inference algorithm is that the annotated type of each function can be assigned to the corresponding annotated function in the type system defined in [7]. The main constraints posed by that system with respect to regions are reflected in the function and constructor typing rules, shown in Fig. 5.

$$\frac{\begin{array}{c}\mathrm{fresh}(\rho_{self}), \quad \rho_{self} \notin \mathit{regions}(s) \quad \mathcal{R} = \mathit{regions}(\overline{t_i}^n) \cup \{\overline{\rho_j}^l\} \cup \mathit{regions}(s) \\ \Gamma + \overline{[x_i : t_i]}^n + \overline{[r_j : \rho_j]}^l + [\mathit{self} : \rho_{self}] + [f : \forall \rho \in \mathcal{R} \,.\, \overline{t_i}^n \to \overline{\rho_j}^l \to s] \vdash e : s\end{array}}{\{\Gamma\} \;\; f \; \overline{x_i}^n \; @ \; \overline{r_j}^l = e \;\; \{\Gamma + [f : \mathit{gen}(\forall \rho \in \mathcal{R} \,.\, \overline{t_i}^n \to \overline{\rho_j}^l \to s, \Gamma)]\}} \; [\mathrm{FUNB}]$$

$$\frac{\Sigma(C) = \sigma \quad \overline{s_i}^n \to \rho \to T \; @\overline{\rho}^m \trianglelefteq \sigma \quad \Gamma = (\overline{[a_i : s_i]}^n_{i=1}) + [r : \rho]}{\Gamma \vdash C \; \overline{a_i}^n @ r : T \; @\overline{\rho}^m} \; [\mathrm{CONS}]$$

Fig. 5. Typing rules for function definition and constructor application

In rule [FUNB] the fresh (local) program region variable *self* is assigned a fresh type variable ρ_{self} that cannot appear in the function result type. This prevents dangling pointers arising by region deallocation at the end of a function call. The only regions in scope for writing are *self* and the argument regions.

Notice that region-polymorphic recursion is allowed: inside the body e, different applications of f may use different regions. We use $gen(\sigma', \Gamma)$ and $tf \trianglelefteq \sigma$ to respectively denote (standard) generalization of a type with respect to type variables excluding region types, and instantiation of a polymorphic type.

The types of the constructors are given in an initial environment Σ built from the datatype declarations. These types reflect the fact that the recursive substructures live in the same region. For example, in the case of lists and trees:

$$\begin{array}{l}
[\,] : \forall a, \rho.\rho \to [a]@\rho \\
(:) : \forall a, \rho.a \to [a]@\rho \to \rho \to [a]@\rho \\
\mathit{Empty} : \forall a, \rho.\rho \to \mathit{Tree}\ a@\rho \\
\mathit{Node} : \forall a, \rho.\mathit{Tree}\ a@\rho \to a \to \mathit{Tree}\ a@\rho \to \rho \to \mathit{Tree}\ a@\rho
\end{array}$$

As a consequence, rule [CONS] may force some of the actual arguments to live in the same regions.

3.1 A High-Level View of the Algorithm

Figure 6 shows a high-level view of the Hindley-Milner (abbreviated HM in the following) type inference algorithm of the *Safe* compiler, written in Haskell, in which some parts have to do with region inference.

The first phase, `decorDecsData`, annotates the **data** declarations with region variables and infers the types of the data constructors. These are saved in the assumption environment `as`. A fresh region variable is generated for each non-recursive nested data type and one more for the type being defined, which is placed as an additional argument of each constructor. Only the recursive occurrences are forced to have the same region arguments. All the region variables are reflected in the type so that all the regions in which the structure has a portion are known. In Sec. 2.2 we have shown some examples of the result produced by this phase.

After this, the equations `defs` defining functions are grouped by function name, traversed, and their HM-types and regions inferred for each function (algorithm `decorAndGenOuterDefs`, see below), accumulating the inferred type in the assumption environment `as` in order to infer subsequent function definitions.

```
decorProg :: Assumps -> Prog a -> (Assumps, Prog ExpTipo)
decorProg asInit (datas, defs, exp) =  (as',(datas', concat defs', exp'))
  where (as,datas')  = decorDecsData asInit datas
        groups       = groupBy sameName defs
        (as', defs') = mapAccumL decorAndGenOuterDefs as groups
        exp'         = decorAndGenMainExp as' exp
```

Fig. 6. A high-level view of the Hindley-Milner inference algorithm

```
$decorAndGenOuterDefs\ \Gamma\ Defs = (\Gamma \cup [f \mapsto t^+], Defs'')$
    where $f$                                          $= extractFunctionName\ Defs$
          $(Defs', Eqs, Fresh_{expl}, \overline{trec_j}^p)$ $= decorAndGenEqs\ \Gamma\ Defs$
          $\theta_1$                                   $= solveEqs\ Eqs$
          $t$                                          $= \theta_1(type\ Defs')$
          $(\theta_2, \overline{\varphi_j}^p)$        $= handleRecCalls\ t\ (\theta_1(\overline{trec_j}^p))$
          $\theta$                                     $= \theta_2 \circ \theta_1$
          $R_{expl}$                                   $= \theta(Fresh_{expl})$
          $(\theta_{self}, t^+, RegMap)$               $= inferRegions\ t\ R_{expl}\ \overline{\varphi_j}^p$
          $Defs''$                                     $= annotateDef\ (\theta_{self} \circ \theta)\ RegMap$
```

Fig. 7. HM-type and region inference for a single function

Finally, the main expression `exp` of the program is inferred, and decorated by `decorAndGenMainExp` (not shown).

3.2 Region Inference of Function Definitions

Figure 7 shows in Haskell-like pseudocode the HM-inference process for a single function consisting of a list *Defs* of equations. Let us call such function f.

We have a decoration phase *decorAndGenEqs* which generates fresh type and region type variables, and equations relating types that have to be unified, but delays all the unifications to a subsequent phase. Some of these equations correspond to the usual HM type inference, e.g. $a = [b] \rightarrow b$, but some other unify region type variables, e.g. $\rho_1 = \rho_2$. The decoration phase generates a set $Fresh_{expl}$ of fresh region type variables assigned to the region arguments of constructor applications and (already inferred) function applications. This set will be needed in the second phase of region inference.

Unification equations are solved by *solveEqs* and *handleRecCalls*. The former solves all the equations in the usual HM style except those related to the recursive applications of f, which are solved in a special way by the latter: Hindley-Milner types of recursive applications are unified with the inferring function's type, while region type variables are not unified. This is due to the fact that the type $trec_j$ of every application of f should be a fresh instance of the HM type t of the function with respect to the region types. Each region substitution φ_j reflects this fact by mapping the region type variables in t to those in $trec_j$. For instance, if the type inferred for a function after *solveEqs* is $[a]@\rho_1 \rightarrow b$ and there is a single recursive application with type $[a]@\rho_2 \rightarrow [c]@\rho_2$, the resulting substitution of *handleRecCalls* is $\theta = [b \mapsto [c]@\rho_1]$ with a region mapping $\varphi = [\rho_1 \mapsto \rho_2]$.

The next step is the application of the final substitution θ to the set $Fresh_{expl}$ of explicit region types obtained above, obtaining the smaller set R_{expl}. Then, the second and final phase, *inferRegions*, of region inference is done. Its purpose is to detect how many explicit region arguments the (possibly recursive) function f must have, and to infer which region types must be assigned to the local working region *self*. This algorithm is depicted in Fig. 8 and explained in the next section. It delivers a substitution θ_{self} mapping some region type variables to the reserved type variable ρ_{self} assigned to the local region *self*, a map *RegMap*

```
$inferRegions\ t\ R_{expl}\ \overline{\varphi_j}^p = ([\rho \mapsto \rho_{self} \mid \rho \in R_{self}], \overline{t_i}^n \to \overline{\rho_k}^m \to t', [\overline{\rho_k \mapsto r_j}^m])$
    where $\overline{t_i}^n \to t' = t$
          $R_{out} = regions\ t'$
          $R_{in} = regions\ \overline{t_i}^n$
          $R_{arg} = R_{expl} \cap (R_{in} \cup R_{out})$
          $(R'_{arg}, R'_{expl}) = computeRargFP\ R_{in}\ R_{out}\ R_{arg}\ R_{expl}\ \overline{\varphi_j}^p$
          $R_{self} = R'_{expl} - (R_{out} \cup R_{in})$
          $\overline{\rho_k}^m = R'_{arg}$

$computeRargFP\ R_{in}\ R_{out}\ R_{arg}\ R_{expl}\ \overline{\varphi_j}^p$
        | $R_{arg} == R'_{arg} = (R'_{arg}, R'_{expl})$
        | otherwise $= computeRargFP\ R_{in}\ R_{out}\ R'_{arg}\ R'_{expl}\ \overline{\varphi_j}^p$
    where $R'_{expl} = R_{expl} \cup \bigcup_{j=1}^{p} \{\varphi_j(\rho) \mid \rho \in R_{arg}\}$
          $R'_{arg} = R'_{expl} \cap (R_{in} \cup R_{out})$
```

Fig. 8. Second phase of the region inference algorithm

mapping some other region type variables to region arguments, and the extended function type t^+. The last step adds these region arguments to the definition of f. The function's body is traversed again and the above substitutions and mappings are used to incorporate the appropriate region arguments to all the expressions, including the recursive applications of f. Additionally, the final substitution $\theta_{self} \circ \theta$ is applied to all the types.

3.3 Second Phase of Region Inference

Algorithm *inferRegions* of Fig. 8 receives the type t obtained for the function f by the HM inference, the set R_{expl} of initial explicit region types, and the list of substitutions $\overline{\varphi_j}^p$ associated with the recursive applications of f. First, it computes the sets R_{in} and R_{out} of region type variables of respectively the argument and the result parts of t. Let ρ_{self} be an additional fresh type variable for *self*.

Given these three sets, the region inference problem can be specified as finding three sets R'_{expl}, R'_{arg} and R_{self}, respectively standing for the sets of final explicit region types, of region types needed as additional arguments of f, and of region types that must be unified with ρ_{self}, subject to the following restrictions:

1. $R'_{expl} \subseteq R_{self} \cup R'_{arg}$
2. $R_{self} \cap R'_{arg} = \emptyset$
3. $R_{self} \cap (R_{in} \cup R_{out}) = \emptyset$
4. Every recursive application of f is typeable

The first one expresses that everything built by f's body must be in regions in scope. The second and third ones state that region *self* is fresh and hence different from any other region received as an argument or where an input argument lives. These restrictions and the extension of (3) to R_{out} are enforced by the typing rule [FUNB]. The last one can be further formalised by requiring that f's type, extended with the region arguments in R'_{arg}, can produce type instances for typing all the recursive applications of f, each one extended with as many region arguments as the cardinal of R'_{arg}. So, in order to satisfy restriction (4) one must provide a decoration of each recursive application of f with appropriate region arguments, of region types belonging either to R_{self} or to R'_{arg}, as restriction (1) requires.

In the extended version of this paper [9] we show that any sets R'_{expl}, R'_{arg} and R_{self} satisfying these restrictions produce a version of f which admits a type in the type system. The correctness of the type system with respect to the semantics was established in [7]. There, we proved that dangling pointers arising from region deallocation or destructive pattern matching are never accessed by a well-typed program.

Notice that an algorithm choosing any $R'_{arg} \supseteq R'_{expl}$ and $R_{self} = \emptyset$ would be correct according to this specification. But this solution would be very poor as, on the one hand no construction would ever be done in the *self* region and, on the other, there might be region arguments never used. We look for an optimal solution in two senses. On the one hand, we want R'_{arg} to be as small as possible, so that only those regions where data are built are given as arguments. On the other hand, we want R_{self} to be as big as possible, so that the maximum amount of memory is deallocated at function termination.

3.4 The Kernel of the Algorithm

Our algorithm initially computes $R_{arg} = R_{expl} \cap (R_{in} \cup R_{out})$, by using the set R_{expl} of initial explicit region types. Then, it starts a fixpoint algorithm *computeRargFP* (see Fig. 8) trying to get the type of f's recursive applications as instances of the type of f extended with the current set R_{arg} of arguments. It may happen that the set of explicit regions R'_{expl} may grow while considering different applications (see the examples in Sec. 4). Adding more explicit variables to one application will influence the type of the applications already inferred. As R'_{arg} depends on R'_{expl}, it may also grow. So, a fixpoint is used in order to obtain the final R'_{arg} and R'_{expl} from the initial ones. Due to our solution above, R'_{arg} cannot grow greater than $R_{in} \cup R_{out}$, so termination of the fixpoint is guaranteed. Once obtained the final R'_{arg} and R'_{expl}, the set R_{self} is computed as $R_{self} = R'_{expl} - (R_{in} \cup R_{out})$. Notice that $R'_{arg} = R'_{expl} \cap (R_{in} \cup R_{out})$ is an invariant of the algorithm.

We show below that these choices maximise the data allocated to the *self* region, which in turn maximises the amount of memory reclaimed at runtime when the corresponding function call finishes. With respect to the remaining DSs not being inferred to live in *self*, they will be allocated to the regions which are parameters to the function being called. It is the *caller* function's responsibility to determine where to put these DSs by passing the suitable arguments. Since the caller function is also inferred by the algorithm, the parameter assignment is done in such a way that the data allocated in the caller's *self* region is also maximised. From a global point of view, every cell not being created in the current topmost region (i.e. the region bound to the *self* identifier) will be created in the highest possible region and hence, will be deallocated at the earliest time allowed by the type system.

3.5 Correctness, Optimality and Efficiency

First we prove that the proposed solution satisfies the above specification:

1. $R'_{expl} \subseteq (R'_{expl} - (R_{in} \cup R_{out})) \cup (R'_{expl} \cap (R_{in} \cup R_{out}))$
2. $(R'_{expl} - (R_{in} \cup R_{out})) \cap (R'_{expl} \cap (R_{in} \cup R_{out})) = \emptyset$
3. $(R'_{expl} - (R_{in} \cup R_{out})) \cap (R_{in} \cup R_{out}) = \emptyset$

The three immediately follow by set algebra. We will show now that it is optimal: let us assume a different solution $\hat{R}_{self}, \hat{R}_{expl}, \hat{R}_{arg}$ satisfying the above restrictions. Notice that $R_{expl} \subseteq R'_{expl}$ by construction. Without loss of generality we can rename those variables in $\hat{R}_{expl}$ which decorate copy expressions, constructor applications and function calls differerent from f, so that such decorations coincide with those in R'_{expl}. After such renaming $R_{expl} \subseteq \hat{R}_{expl}$. We can also rename the argument regions in recursive calls to f that also appear in R'_{expl}. For example, assume there is a recursive call decorated by R'_{expl} as $f :: \overline{t_i}^n \to \rho'_1 \to \rho'_2 \to t$. If that recursive call was decorated by $\hat{R}_{expl}$ as $f :: \overline{t'_i}^n \to \hat{\rho_1} \to \hat{\rho_2} \to \hat{\rho_3} \to t'$, then $\hat{\rho_1}$ would be renamed as ρ'_1 and $\hat{\rho_2}$ as ρ'_2.

We must show that $\hat{R}_{self} \subseteq R_{self}$ and $R'_{arg} \subseteq \hat{R}_{arg}$. Let us assume $\rho \in R'_{arg}$. By definition of R'_{arg}, $\rho \in R'_{expl}$ and $\rho \in R_{in} \cup R_{out}$. By (3), $\rho \in R_{in} \cup R_{out}$ implies that $\rho \notin \hat{R}_{self}$. Now we distinguish two cases:

$\rho \in R_{expl}$ As $R_{expl} \subseteq \hat{R}_{expl}$, then $\rho \in \hat{R}_{expl}$. By (1) $\rho \in \hat{R}_{arg}$.

$\rho \in R'_{expl} - R_{expl}$ If $\rho \in \hat{R}_{expl}$, then by $\rho \in \hat{R}_{arg}$. Otherwise, R'_{expl} contains more explicit variables which are also arguments of f than $\hat{R}_{expl}$. This case is not possible because R'_{expl} is the least fixpoint of function *computeRargFP* by construction. By (4), $\hat{R}_{expl}$ is also a fixpoint of *computeRargFP*; otherwise, the recursive calls would not be typeable.

Consequently, $\rho \in R'_{arg}$. So, R_{arg} is as small as possible. By constraints (2) and (1), then R_{self} is as big as possible. Regarding regions, there are no principal types in our system, since other correct types bigger than our minimal type could not be obtained as an instance of it.

Our sets are implemented as balanced trees, and operations such as $\cup$, $\cap$, and '$-$' are done in a time in $\Theta(n + m)$, being n and m the cardinalities of the respective sets, so each iteration of the fixpoint algorithm is linear with the number of region type variables occurring in a function body. As it is done in [10], considering as the term size n the sum of the sizes of the abstract syntax tree and of the HM type annotations, each iteration needs time linear with this size. If several iterations are needed, these cannot be more than the number of region type variables in $R_{in} \cup R_{out}$. This gives us $O(n^2)$ cost in the worst case.

4 Examples

As a first example, consider the previously defined function *partition*. A region variable ρ_1 is created for the input list, so that it has type $[Int]@\rho_1$. In addition seven fresh type region variables are generated, one for each constructor application, let say ρ_2 to ρ_8, and so $\mathit{Fresh}_{expl} = \{\rho_2, \ldots, \rho_8\}$. We show them as annotations in the program just in order to better explain the example:

$$\begin{array}{ll} \mathit{partition}\ y\ [\,] = ([\,] :: \rho_2, [\,] :: \rho_3) :: \rho_4 & \\ \mathit{partition}\ y\ (x : xs) & \mid x \le y = (x : ls :: \rho_5, gs) :: \rho_6 \\ & \mid x > y = (ls, x : gs :: \rho_7) :: \rho_8 \\ & \mathbf{where}(ls, gs) = \mathit{partition}\ y\ xs \end{array}$$

The type inference rules generate the following equations relative to these type region variables: $\rho_2 = \rho_5$, $\rho_3 = \rho_7$, and $\rho_4 = \rho_6 = \rho_8$, so the initial R_{expl} in this case is $\{\rho_2, \rho_3, \rho_4\}$. After unification, the type of partition is $\mathit{Int} \to [\mathit{Int}]@\rho_1 \to ([\mathit{Int}]@\rho_2, [\mathit{Int}]@\rho_3)@\rho_4$, so $R_{in} = \{\rho_1\}$ and $R_{out} = \{\rho_2, \rho_3, \rho_4\}$. Then, $R_{arg} = \{\rho_2, \rho_3, \rho_4\}$. Now we shall compare the type of the definition (augmented with the variables of R_{arg}) and the type used in the recursive call, where the tuple (ls, gs) is assumed to live in the region ρ_9.

$$\begin{array}{r} \text{Definition: } \mathit{Int} \to [\mathit{Int}]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([\mathit{Int}]@\rho_2, [\mathit{Int}]@\rho_3)@\rho_4 \\ \text{Rec. call: } \mathit{Int} \to [\mathit{Int}]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_9 \to ([\mathit{Int}]@\rho_2, [\mathit{Int}]@\rho_3)@\rho_9 \end{array}$$

We obtain the region substitution $\varphi = [\rho_1 \mapsto \rho_1, \rho_2 \mapsto \rho_2, \rho_3 \mapsto \rho_3, \rho_4 \mapsto \rho_9]$. As a consequence, the variable ρ_9 is made explicit, so $R_{expl} = \{\rho_2, \rho_3, \rho_4, \rho_9\}$. The set R_{arg} does not change and hence the fixpoint has been computed. We get $R_{self} = \{\rho_9\}$ and the program is annotated as follows:

$$\begin{array}{ll} \mathit{partition} :: \mathit{Int} \to [\mathit{Int}]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([\mathit{Int}]@\rho_2, [\mathit{Int}]@\rho_3)@\rho_4 & \\ \mathit{partition}\ y\ [\,]\ @\ r_2\ r_3\ r_4 = ([\,]@r_2, [\,]@r_3)@r_4 & \\ \mathit{partition}\ y\ (x : xs)\ @\ r_2\ r_3\ r_4 & \mid x \le y = ((x : ls)@r_2, gs)@r_4 \\ & \mid x > y = (ls, (x : gs)@r_3)@r_4 \\ & \mathbf{where}\ (ls, gs) = \mathit{partition}\ y\ xs\ @\ r_2\ r_3\ \mathit{self} \end{array}$$

Notice that the tuple resulting from the recursive call to *partition* is located in the working region. Without region-polymorphic recursion this tuple would have to be stored in the output region r_4, requiring $O(n)$ space in a caller region.

Another example is the dynamic programming approach to computing binomial coefficients by using the Pascal's triangle. We start from the unit list [1], which corresponds to the 0-th row of the triangle. If $[x_0, x_1, \ldots, x_{i-1}, x_i]$ are the elements located on the i-th row, then the elements of the $i+1$-th row are given by the list $[x_0 + x_1, x_1 + x_2, \ldots, x_{i-1} + x_i, x_i]$. The binomial coefficient $\binom{n}{m}$ can be obtained from the m-th element in the n-th row of the Pascal's triangle. Function *sumList*, computes the $i+1$-th row of the triangle from its i-th row:

$$\begin{array}{l} \mathit{sumList}\ (x : [\,])\ = (x : [\,] :: \rho_2) :: \rho_3 \\ \mathit{sumList}\ (x : xs) = (x + y : \mathit{sumList}\ xs) :: \rho_4 \quad \mathbf{where}\ (y : \_) = xs \end{array}$$

In the definition above we just show those region variables belonging to Fresh_{expl}. Let us assume that after unification the input list has type $[\mathit{Int}]@\rho_1$. In addition, the variables ρ_2, ρ_3 and ρ_4 are unified, so $R_{expl} = \{\rho_2\}$ and the inferred type (without region parameters) for *sumList* is $[\mathit{Int}]@\rho_1 \to [\mathit{Int}]@\rho_2$. Hence we get $R_{in} = \{\rho_1\}$, $R_{out} = \{\rho_2\}$ and $R_{arg} = \{\rho_2\}$. We extend the signature of *sumList* to $[\mathit{Int}]@\rho_1 \to \rho_2 \to [\mathit{Int}]@\rho_2$.

Next we analyse the recursive call. Since the inferred type for xs is $[Int]@\rho_1$ and the type of the recursive call's result is $[Int]@\rho_2$, the type for *sumList* in this call is $[Int]@\rho_1 \to [Int]@\rho_2$. By pairing with the type of *sumList* in the definition, we get that the additional argument needed in the recursive call also has type ρ_2. Therefore, ρ_2 is added to the set of explicit variables R_{expl}. Since it was already in this set, R_{expl} stays the same as the one calculated previously and hence R_{arg} also does, so the fixpoint has been reached. Finally we obtain $R_{self} = \emptyset$ and the function is annotated as follows:

$$\begin{array}{l}
\mathit{sumList} \quad :: \ [Int]@\rho_1 \to \rho_2 \to [Int]@\rho_2 \\
\mathit{sumList}\ (x : [\,])\ @\ r = (x : ([\,]\ @\ r))@\ r \\
\mathit{sumList}\ (x : xs)\ @\ r = (x + y : \mathit{sumList}\ xs\ @\ r)\ @\ r \quad \mathbf{where}\ (y : \_) = xs
\end{array}$$

Function *pascal* iterates over the initial list in order to get the desired row. Below we show the region variables generated in constructor applications and in non-recursive function applications, just *after* type unification:

$$\begin{array}{l}
\mathit{pascal}\ 0 = (1 : [\,] :: \rho_1) :: \rho_1 \\
\mathit{pascal}\ n = (1 : \mathit{sumList}\ (\mathit{pascal}\ (n-1))) :: \rho_1
\end{array}$$

The type inferred for *pascal* is $Int \to [Int]@\rho_1$. Hence $R_{in} = \emptyset$, $R_{out} = \{\rho_1\}$ and $R_{expl} = \{\rho_1\}$, which gives us an initial $R_{arg} = \{\rho_1\}$. The type signature for *pascal* changes accordingly to $Int \to \rho_1 \to [Int]@\rho_1$. Let us assume that the result of the recursive call to *pascal* has type $[Int]@\rho_2$. Therefore, the type of this function in the recursive call is $Int \to \rho_2 \to [Int]@\rho_2$. Since ρ_2 is now made explicit, it is added to R_{expl}, which now contains the region variables $\{\rho_1, \rho_2\}$. However, R_{arg} stays the same and hence the fixpoint has been reached. Finally, we get $R_{self} = \{\rho_2\}$ and the program is annotated as follows:

$$\begin{array}{l}
\mathit{pascal} \quad :: \ Int \to \rho_1 \to [Int]@\rho_1 \\
\mathit{pascal}\ 0\ @\ r = (1 : [\,]\ @\ r)@\ r \\
\mathit{pascal}\ n\ @\ r = (1 : \mathit{sumList}\ (\mathit{pascal}\ (n-1)\ @\ \mathit{self})\ @\ r)\ @\ r
\end{array}$$

The resulting list from the recursive call to *pascal* will be destroyed once the calling function finishes. Hence a function call *pascal* n has a cost of $O(n)$ in space. Without region-polymorphic recursion the result of every recursive call would be built in the output region r, which would imply $O(n^2)$ heap cost.

5 Related Work and Conclusions

The pioneer work on region inference is that of M. Tofte, J.-P. Talpin and their colleagues on the MLKit compiler [14,10] (in what follows, TT). Their language is higher-order and they also support polymorphic recursion in region arguments. The TT algorithm has two phases, respectively called S and R. The S-algorithm just generates fresh region variables for values and introduces the lexical scope of the regions by using a **letregion** construct. The R-algorithm is responsible for assigning types to recursive functions. It deals with region-polymorphic recursion and also computes a fixpoint. The total cost is in $O(n^4)$. The meaning of a typed expression **letregion** ρ **in** $e : \mu$ is that region ρ does not occur free in type μ, so it can be deallocated upon the evaluation of e. Our algorithm has some resemblances with this part of the inference, in the sense that we decide to unify with ρ_{self} all the region variables not occurring in the result type of a function. They do not claim their algorithm to be optimal but in fact they create as many regions as possible, trying to make local *all* the regions not needed in the final value. One problem reported in [12] is that most of the regions inferred in the first versions of the algorithm contained a single value so that region management produced a big overhead at runtime. Later, they added a new analysis to collapse all these regions into a single one local to the invocation (allocated in the stack). So, having a single local region *self* per function invocation does not seem to us to be a big drawback if function bodies are small enough. We believe that region-polymorphic recursion has a much bigger impact in avoiding memory leaks than multiplicity of local regions. So, we claim that the results of our algorithm are comparable to those of TT for first-order programs.

A radical deviation from these approaches is [4] which introduces a type system in which region life-times are not necessarily nested. The compiler annotates the program with region variables and supports operations for allocation, releasing, aliasing and renaming. A reference-counting analysis is used in order to decide when a released region should be deallocated. The language is first-order. The inference algorithm [6] can be defined as a global abstract interpretation of the program by following the control flow of the functions in a backwards direction. Although the authors do not give either asymptotic costs or actual benchmarks, it can be deduced that this cost could grow more than quadratically with the program text size in the worst case, as a global fixpoint must be computed and a region variable may disappear at each iteration. This lack of modularity could make the approach unpractical for large programs.

Another approach is [3] in which type-safe primitives are defined for creating, accessing and destroying regions. These are not restricted to have nested lifetimes. Programs are written and manually typed in a C-like language called *Cyclone*, then translated to a variant of λ-calculus, and then type-checked. So, the price of this flexibility is having explicit region control in the language.

The main virtue of our design is its simplicity. The previous works have no restrictions on the placement of cells belonging to the same data structure. Also, in the case of TT and its derivatives, they support higher-order functions. As a consequence, the inference algorithms are more complex and costly. In our language, regions also suffer from the nested lifetimes constraint, since both region allocation and deallocation are bound to function calls. However, the destructive pattern matching facility compensates for this, since it is possible to dispose of a data structure without deallocating the whole region where it lives. Allocation and destruction are not necessarily nested, and our type system protects the programmer against misuses of this feature. Since allocation is implicit, the price of this flexibility is the explicit deallocation of cells.

In the near future we plan to extend *Safe* to support higher-order functions and mutually recursive data structures. We expect high difficulties in other aspects of the language such as extending dangling pointers safety analyses or memory bounds inference, but not so many to extend the region inference algorithm presented here. It is still open whether we could achieve a cost better than the $O(n^4)$ got by Tofte and Talpin.

References

1. Aiken, A., Fähndrich, M., Levien, R.: Better static memory management: improving region-based analysis of higher-order languages. In: ACM SIGPLAN Conference on Programming Languages Design and Implementation, PLDI 1995, pp. 174–185. ACM Press, New York (1995)
2. Birkedal, L., Tofte, M., Vejlstrup, M.: From region inference to von Neumann machines via region representation inference. In: 23rd ACM Symposium on Principles of Programming Languages, POPL 1996, pp. 171–183. ACM Press, New York (1996)
3. Fluet, M., Morrisett, G., Ahmed, A.: Linear regions are all you need. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 7–21. Springer, Heidelberg (2006)
4. Henglein, F., Makholm, H., Niss, H.: A direct approach to control-flow sensitive region-based memory management. In: 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP 2001, pp. 175–186. ACM Press, New York (2001)
5. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: 30th ACM Symposium on Principles of Programming Languages, POPL 2003, pp. 185–197. ACM Press, New York (2003)
6. Makholm, H.: A language-independent framework for region inference. Ph.D thesis, Univ. of Copenhagen, Dep. of Computer Science, Denmark (2003)
7. Montenegro, M., Peña, R., Segura, C.: A type system for safe memory management and its proof of correctness. In: 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP 2008, pp. 152–162 (2008)
8. Montenegro, M., Peña, R., Segura, C.: An inference algorithm for guaranteeing safe destruction. In: Hanus, M. (ed.) Logic-Based Program Synthesis and Transformation. LNCS, vol. 5438, pp. 135–151. Springer, Heidelberg (2009)
9. Montenegro, M., Peña, R., Segura, C.: A simple region inference algorithm for a first-order functional language (extended version). Technical report, SIC-5-09. Dpto. de Sist. Informáticos y Computación. UCM (2009), http://federwin.sip.ucm.es/sic/investigacion/publicaciones/informes-tecnicos
10. Tofte, M., Birkedal, L.: A region inference algorithm. ACM Transactions on Programming Languages and Systems 20(4), 724–767 (1998)
11. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T.H., Sestoft, P.: Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark (2006)
12. Tofte, M., Hallenberg, N.: Region-Based Memory Management in Perspective. In: Invited talk Space 2001 Work, London, January 2001, pp. 1–8. Imperial College (2001)
13. Tofte, M., Talpin, J.-P.: Implementing the call-by-value lambda-calculus using a stack of regions. In: 21st ACM Symposium on Principles of Programming Languages, POPL 1994, January 1994, pp. 188–201 (1994)
14. Tofte, M., Talpin, J.-P.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)

A Theoretical Framework for the Declarative Debugging of Functional Logic Programs with Lambda Abstractions*

Rafael del Vado Vírseda and Ignacio Castiñeiras

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
rdelvado@sip.ucm.es, ncasti@fdi.ucm.es

Abstract. In this paper, we extend the well-known Naish's declarative debugging scheme for diagnosing wrong computed answers in first-order lazy functional-logic programs to the higher-order setting of the simply typed λ-calculus, where programs are presented by conditional pattern rewrite systems. Our approach generalizes and combines declarative debugging techniques previously developed for less expressive declarative programming paradigms involving applicative rewrite rules instead of λ-abstractions and decidable higher-order unification. Debugging starts with the observation of a wrong computed answer which the user regards as incorrect w.r.t. an intended model that provides a declarative description of the program's semantics. Debugging proceeds by exploring an abridged proof tree built on a higher-order rewriting logic with λ-abstractions that provides a purely declarative view of the computation. Finally, debugging ends with the detection of a defined function rule in the program that is incorrect w.r.t. the intended model. We prove the logical correctness of the debugging method for any sound goal solving system whose computed answers are logical consequences of the program.

1 Introduction

According to a well-known conception, programs in a declarative programming language can be viewed as theories in some suitable logic, while computations can be viewed as deductions. The *Constructor-based ReWriting Logic* CRWL [5] provides a suitable framework for rule-based declarative (functional and logic) programming with non-deterministic and lazy functions with call-time choice semantics, where programs are constructor-based *Conditional Term Rewrite Systems* (CTRS for short). As a concrete example, the following "Prolog-like" CTRS fragment defines a possibly non-deterministic function *flight*, given by first-order

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS), TIN2008-06622-C03-01 (FAST-STAMP), S-0505/TIC/0407 (PROMESAS-CAM), and UCM-BSCH-GR58/08-910502 (GPD-UCM).

conditional rewrite rules (→) where the conditional part (formed only by equations ==) is delimited by the "⇐" symbol:

$$\begin{array}{llllll}
\mathit{connection}\,(\mathit{madrid}, \mathit{tokyo}) & \to & \mathit{false} & & & \\
\mathit{flight}\,(\mathit{valencia}) & \to & \mathit{barcelona} & & & \\
\mathit{flight}\,(\mathit{barcelona}) & \to & X & \Leftarrow & \mathit{connection}\,(X, \mathit{tokyo}) & == \ \mathit{false} \\
\mathit{flight}\,(Y) & \to & X & \Leftarrow & \mathit{flight}\,(X) & == \ Y
\end{array}$$

Functional-logic languages with a sound and complete operational semantics are mainly based on *narrowing*, a transformation rule which combines the basic execution mechanism of functional and logic languages, namely *rewriting* with *unification*. For example, the function *flight* is non-deterministic because the execution by narrowing of the goal $\mathit{flight}\,(\mathit{barcelona}) == F$ yields $\{F \mapsto \mathit{madrid}\}$ and $\{F \mapsto \mathit{valencia}\}$ as computed answers.

Since the classical notion of rewriting is not suitable in this setting, a new notion of rewriting is adopted as the basis of *proof calculi* for joinability (==) and reduction (→) statements. The most important result is the existence of sound and complete *lazy narrowing calculi* [5,18] for solving goals in first-order CRWL-theories presented by CTRS-programs. Moreover, a higher-order extension of CRWL is presented in [4] but using only *applicative rewrite rules* instead of λ-abstractions and higher-order unification.

In this paper, we use a higher-order rewriting logic (called GHRC, *Goal-oriented Higher-order Rewriting Calculus*) for declarative programming with higher-order functions and λ-terms as data structures to obtain more of the expressivity of higher-order functional programming. More precisely, we adopt the framework of the simply typed λ-calculus in which terms are in $\beta\eta$-normal form and theories are presented by *Conditional Pattern Rewrite Systems* (CPRS for short). As a simple example of such higher-order programs, the following pattern rewrite system (adapted from [7]) shows how simple circuits and hardware gates can be represented and computed within higher-order functional-logic programming with lambda abstractions.

$$\begin{array}{lll}
\mathit{map}\,(\lambda u, v.\, F(u,v), [\,]) & \to & [\,] \\
\mathit{map}\,(\lambda u, v.\, F(u,v), [\,(X,Y) \mid \mathit{Rs}\,]) & \to & [\,F(X,Y) \mid \mathit{map}\,(\lambda u, v.\, F(u,v), \mathit{Rs})\,]
\end{array}$$

$$\begin{array}{lllllll}
\mathit{nand}\,(0, X) & \to & 1 & \quad & \mathit{size}\,(\lambda u, v.\, u) & \to & 0 \\
\mathit{nand}\,(X, 0) & \to & 1 & & \mathit{size}\,(\lambda u, v.\, v) & \to & 0 \\
\mathit{nand}\,(1, 1) & \to & 0 & & \mathit{size}\,(\lambda u, v.\, \mathit{nand}\,(F(u,v), G(u,v))) & \to & 1 + \\
 & & & & \multicolumn{3}{r}{\mathit{size}\,(\lambda u, v.\, F(u,v)) + \mathit{size}\,(\lambda u, v.\, G(u,v))}
\end{array}$$

The goal of this example is to compute functions composed of *nand*-functions (or gates). Thus we first specify the *nand*-function and some auxiliary functions, as the classical higher-order function *map* on pairs of numbers. In this program, the function *size* serves two purposes. First, it counts the number of *nand* functions, but also assures that some λ-term contains no other functions. For instance, we can synthesize an *or*-function consisting of three *nand*-gates from the following goal composed by two equality statements.

$$\lambda x, y.\, map\,(F, [\,(0,0), (x,1), (1,y)\,]) == [0,1,1],\; size\,(F) == 3$$

The first equality asserts the behavior of the *or*-function and the second equality specifies the size to restrict the search space. The solution is found by exhaustive search using a *higher-order lazy narrowing calculus with definitional trees* [20]:

$$\{F \mapsto \lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v))\}$$

The previous program is a non-conditional pattern rewrite system, where equality statements are only used in the goal. As a simple example of a conditional pattern rewrite system we can define a higher-order function *diff*, where $diff\,(f,x)$ computes the differential of a function f at some point x:

$$\begin{array}{lcll} diff\,(\lambda y.\, y, x) & \to & 1 & \\ diff\,(\lambda y.\, sin(f(y)), x) & \to & cos(f(x)) * diff\,(\lambda y.\, f(y), x) & \Leftarrow \pi/4 \leq f(x) \leq \pi/2 \\ diff\,(\lambda y.\, ln(f(y)), x) & \to & diff\,(\lambda y.\, f(y), x)/f(x) & \Leftarrow f(x) \neq 0 \end{array}$$

In this work, only equality statements are supported by our current programming framework, although the same ideas can be applied to integrate non-equality constraints in the conditional part of pattern rewrite rules and goals.

We are interested in the logical characterization and the practical application to debugging of the semantics of programs formalized by *constructor-based* CPRSs, where the notion of lazy and possibly non-deterministic higher-order functions and conditional equations involving λ-abstractions plays a central role. In contrast to more traditional frameworks such as *equational logic* and alternative approaches such as *needed rewriting* [7] the higher-order rewriting logic on lambda abstractions GHRC has the ability to characterize the intended computational behavior based on *conditional higher-order narrowing* for non-determinism in a correct and efficient way [5,18,20] (see Section 7 for more related work and comparisons with other approaches).

Recently, in [21] we have presented a model-theoretic semantics from traditional theories in higher-order declarative (functional and logic) programming, and a fixpoint semantics that matches the pattern model of a CPRS, useful for verification and algorithmic debugging purposes. However, in this paper we prefer to develop an independent and more easy theoretical framework for declarative debugging based on the notion of *interpretation* of a CPRS, in order to generalize and combine declarative debugging techniques and implementations previously developed for first-order lazy functional-logic programming [1,2].

2 Motivating Example

A frequent claim about declarative programming languages is that the task of reasoning about programs (as e.g., CTRSs or CPRSs) is easier than in other programming paradigms because of the existence of an underlying logic providing

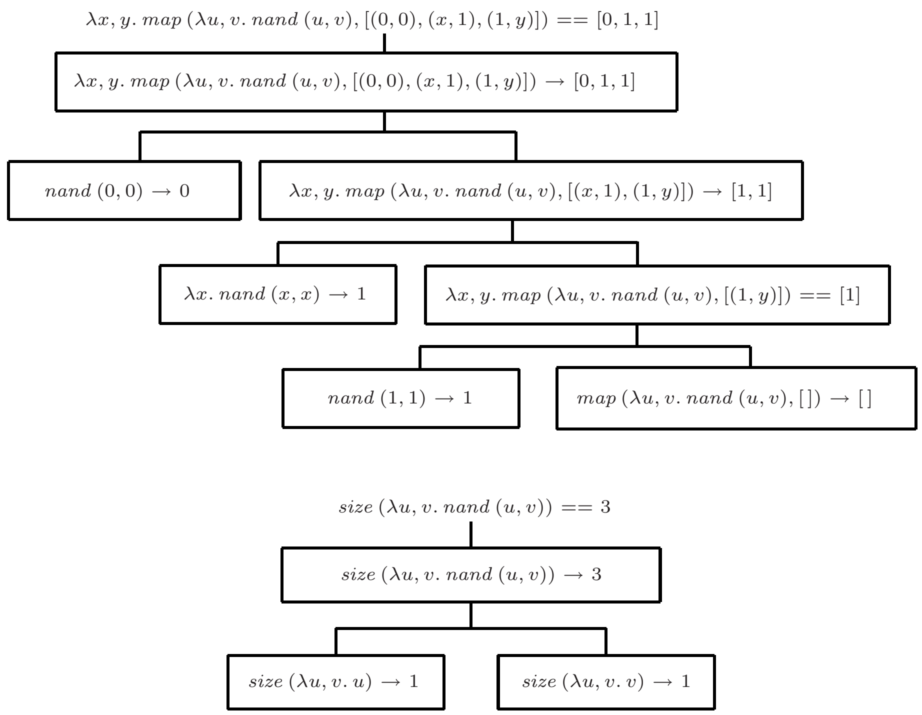

λx, y. map (λu, v. nand (u, v), [(0, 0), (x, 1), (1, y)]) == [0, 1, 1]
λx, y. map (λu, v. nand (u, v), [(0, 0), (x, 1), (1, y)]) → [0, 1, 1]
nand (0, 0) → 0
λx, y. map (λu, v. nand (u, v), [(x, 1), (1, y)]) → [1, 1]
λx. nand (x, x) → 1
λx, y. map (λu, v. nand (u, v), [(1, y)]) == [1]
nand (1, 1) → 1
map (λu, v. nand (u, v), [ ]) → [ ]
size (λu, v. nand (u, v)) == 3
size (λu, v. nand (u, v)) → 3
size (λu, v. u) → 1
size (λu, v. v) → 1

Fig. 1. Computation trees for declarative debugging involving lambda abstractions

more or less natural logical methods for that purpose. In the case of higher-order functional-logic programming, the proof calculus offered by our GHRC approach gives an attractive and mathematically well-founded basis for reasoning on the semantics of programs. In particular, GHRC provides firm theoretical foundations for the declarative debugging of functional-logic programs with lambda abstractions, following the classical declarative debugging approach proposed by Shapiro and Naish [16]. In order to illustrate the main features of this diagnosis technique and to motivate the approach presented in this work we consider a simple debugging example. The following higher-order functional-logic program involving lambda abstractions is an erroneous fragment of the previous pattern rewrite system for hardware synthesis (errors are marked by a box):

$$\begin{array}{lcl} map\,(\lambda u, v.\, F(u,v), [\,]) & \to & [\,] \\ map\,(\lambda u, v.\, F(u,v), [\,(X,Y) \,|\, Rs\,]) & \to & [\,F(X, \boxed{X}) \,|\, map\,(\lambda u, v.\, F(u,v), Rs)\,] \end{array}$$

$$\begin{array}{lclcl} nand\,(0, \boxed{0}) & \to & \boxed{0} & size\,(\lambda u, v.\, u) & \to \ \boxed{1} \\ nand\,(X, \boxed{X}) & \to & 1 & size\,(\lambda u, v.\, v) & \to \ \boxed{1} \\ nand\,(\,1,\,1) & \to & \boxed{1} & size\,(\lambda u, v.\, nand\,(F(u,v), G(u,v))) & \to \ 1\,+ \\ & & & \multicolumn{2}{r}{size\,(\lambda u, v.\, F(u,v)) + size\,(\lambda u, v.\, G(u,v))} \end{array}$$

The debugging technique starts with the observation of a solution computed from a goal and a CPRS by means of a suitable goal solving system (see, e.g., [7,20]). For instance, we consider our previous goal to compute the *or*-function consisting of three *nand*-gates, but now we obtain $\{F \mapsto \lambda u, v.\, \mathit{nand}\,(u, v)\}$ as the computed answer. The user regards this solution as incorrect (because the user really expects as solution $\{F \mapsto \lambda u, v.\, \mathit{nand}\,(\mathit{nand}\,(u, u), \mathit{nand}\,(v, v))\}$) according to their own *intended interpretation* of the declarative description of the program's semantics.

Then, debugging proceeds by exploring a suitable *computation tree*, obtained as a proof tree in the logical calculus offered by the higher-order rewriting logic GHRC for the witness that the obtained computed answer is a solution of the initial goal. This proof tree provides a purely declarative view of the computation, so that the user does not need to understand the complex underlying operational mechanism based on conditional higher-order narrowing described in [7,20]. The computation trees for our current example are graphically represented in Fig. 1 (more on its structure and construction will be explained in Section 6). Each node of this tree represents the computation of some observable result, depending on the results of its children nodes. Declarative diagnosis explores this proof tree looking for a so-called *buggy node* which computes an incorrect result from children whose results are correct; such a node must point to an incorrect program fragment. The search for a buggy node can be implemented with the help of an external *oracle* (usually the user with some semi-automatic support) who has a reliable declarative knowledge of the expected program semantics. Finally, debugging ends with the detection of a function rule in the CPRS $\mathcal{R}$ that is incorrect w.r.t. the intended interpretation $\mathcal{I}$.

For instance, the first computation tree depicted in Fig. 1 has a buggy node because 0 (resp. 1) is not the truth value for $\mathit{nand}\,(0, 0)$ (resp. $\mathit{nand}\,(1, 1)$). Analogously, $\lambda x.\, \mathit{nand}\,(x, x) \to 1$ is incorrect w.r.t. the user's intended program semantics; as we have shown previously, $\mathit{nand}\,(0, 0) \to 1$ and $\mathit{nand}\,(1, 1) \to 0$. After these corrections, there is another buggy node for the function *map* because the user knows that $\mathit{map}\,(\lambda u, v.\, \mathit{nand}\,(u, v), [(1, 1)])$ is $[0]$ instead of $[1]$ as the reduction statement $\lambda y.\, \mathit{map}\,(\lambda u, v.\, \mathit{nand}\,(u, v), [(1, y)]) \to [1]$ claims. Finally, the first and second pattern rewrite rules of the function *size* are also incorrect w.r.t. the user's intended interpretation because $\mathit{size}\,(\lambda u, v.\, u)$ (resp. $\mathit{size}\,(\lambda u, v.\, v)$) yields 0 instead of 1. After these new corrections in the program, no more wrong computed answers will be observed for the goal discussed above, and the right solution $\{F \mapsto \lambda u, v.\, \mathit{nand}\,(\mathit{nand}\,(u, u), \mathit{nand}\,(v, v))\}$ is then obtained.

This paper is structured as follows. In Section 3 we introduce the basic notions and notations from the λ-calculus and higher-order term rewriting which are needed to understand the theoretical framework. In Section 4 we introduce the higher-order conditional rewriting logic characterized by the proof system GHRC, as a generalization of the proof system which underlies the first-order rewriting logic CRWL. Section 5 is concerned with the declarative semantics of GHRC-programs, presented as a simpler alternative to the model-theoretic semantics of [21]. In Section 6 we discuss the application of GHRC to the development of a declarative debugging technique of wrong computed answers for functional-logic programming with lambda abstractions. Finally, Section 7 summarizes some conclusions and presents a brief outline of related and planned future work.

3 Preliminary Notions

We assume the reader is familiar with the notions and notations pertaining to λ-calculus and higher-order term rewriting (see, e.g., [7]). The set of types for simply typed λ-terms is generated by a set $\mathcal{B}$ of *base types* (e.g., `nat`, `bool`) and the function type constructor "$\rightarrow$". Simply typed λ-*terms* are generated in the usual way from a signature $\mathcal{F}$ of *function symbols* and a countably infinite set $\mathcal{V}$ of *variables* by successive operations of abstraction and application. We also consider the enhanced signature $\mathcal{F}_\perp = \mathcal{F} \cup \mathtt{Bot}$, where $\mathtt{Bot} = \{\perp_b \mid b \in \mathcal{B}\}$ is a set of distinguished $\mathcal{B}$-typed constants. The constant $\perp_b$ is intended to denote an *undefined value* of type b. We employ $\perp$ as a generic notation for a constant from `Bot`. In this paper, we assume the following conventions of notation: X, Y, Z, R, H, possibly primed or with subscripts, denote free variables; f, f' denote function symbols, and a a (free or bound) variable or a constant from $\mathcal{F}$; l, r, s, t, u, possibly primed or with subscript, denote terms; $\pi, \pi', \pi_1, \pi_2, \ldots$ denote terms of base type. We also define the *arity* of $f \in \mathcal{F}$ as $ar(f) = n \geq 0$. A sequence of syntactic objects $o_1, \ldots, o_n$, where $n \geq 0$, is abbreviated by $\overline{o_n}$. For instance, the simply typed λ-term $\lambda x_1. \ldots \lambda x_k.(\cdots(a\ t_1)\ \cdots\ t_n)$ is abbreviated by $\lambda\overline{x_k}.a(\overline{t_n})$. Substitutions $\gamma \in \mathit{Subst}(\mathcal{F}_\perp, \mathcal{V})$ are finite type-preserving mappings from variables to terms, denoted by $\{\overline{X_n \mapsto t_n}\}$, and extend homomorphically from terms to terms. By convention, we write ε for the *identity substitution*, $t\gamma$ instead of $\gamma(t)$, and $\gamma\gamma'$ for the function composition $\gamma' \circ \gamma$.

The long $\beta\eta$-normal form of a term, denoted by $t\updownarrow^\eta_\beta$, is the η-expanded form of the β-normal form of t. It is well-known that $s =_{\alpha\beta\eta} t$ if $s\updownarrow^\eta_\beta =_\alpha t\updownarrow^\eta_\beta$ [8]. Since $\beta\eta$-normal forms are always defined, we will in general assume that terms are in long $\beta\eta$-normal form and are identified modulo α-conversion. For brevity, we may write variables and constants from $\mathcal{F}$ in η-normal form, e.g., X instead of $\lambda\overline{x_k}.X(\overline{x_k})$. We assume that the transformation into long $\beta\eta$-normal form is an implicit operation, e.g., when applying a substitution to a term. With these conventions, every term t has a unique long $\beta\eta$-normal form $\lambda\overline{x_k}.a(\overline{t_n})$, where $a \in \mathcal{F}_\perp \cup \mathcal{V}$ and $a()$ coincides with a. The symbol a is called the *root* of t and is denoted by $hd(t)$. We distinguish between the set $\mathcal{T}(\mathcal{F}_\perp, \mathcal{V})$ of *partial* terms (*terms* for short) and the set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of *total* terms. $\mathcal{T}(\mathcal{F}_\perp, \mathcal{V})$ is a poset with respect to the *approximation ordering* $\sqsubseteq$, defined as the least partial ordering such that:

$$\lambda\overline{x_k}.\perp \sqsubseteq \lambda\overline{x_k}.t \qquad t \sqsubseteq t \qquad \frac{s_1 \sqsubseteq t_1 \quad \cdots \quad s_n \sqsubseteq t_n}{\lambda\overline{x_k}.a(\overline{s_n}) \sqsubseteq \lambda\overline{x_k}.a(\overline{t_n})}$$

We adopt the convention that the free and bound variables inside a term are kept disjoint, and assume that bound variables with different binders have different names. The set of free variables of a term t is denoted by $\mathcal{FV}(t)$. To manipulate terms, we define:

- *The set of positions in* t: $Pos(\lambda\overline{x_k}.a(\overline{t_n})) = \{1^i \mid 0 \leq i \leq k\} \cup \{1^k.j.q \mid 1 \leq j \leq n,\ q \in Pos(t_j)\}$, where "." denotes sequence concatenation and 1^k is the sequence of 1 repeated k times. The empty sequence is denoted by ϵ. Note that, with this convention, we have $1^0 = \epsilon$.

- *The subterm* $t|_p$ *of* t *at some position* $p \in Pos(t)$:
$$(\lambda\overline{x_k}.a(\overline{t_n}))|_p = \begin{cases} \lambda x_{i+1} \ldots x_k.a(\overline{t_n}) & \text{if } p = 1^i \text{ with } 0 \leq i \leq k, \\ t_i|_q & \text{if } p = 1^k.i.q \text{ and } 1 \leq i \leq n. \end{cases}$$
A position p is *maximal* in t if $t|_p$ is of base type. The set of maximal positions in a term t is denoted by $MPos(t)$.

- *The sequence of variables abstracted on the path to position* $p \in Pos(t)$:
$$seq_{bv}(t,p) = \begin{cases} \epsilon & \text{if } p = \epsilon, \\ x.seq_{bv}(s,q) & \text{if } t = \lambda x.s \text{ and } p = 1.q, \\ seq_{bv}(t_i,q) & \text{if } t = a(\overline{t_n}),\ 0 < i \leq n, \text{ and } p = i.q. \end{cases}$$
The set of *variables abstracted* on the path to position $p \in Pos(t)$ is $\mathcal{BV}(t,p) = \{seq_{bv}(t,p)\}$, and the set of *variables with bound occurrences* in t is $\mathcal{BV}(t) = \bigcup_{p \in Pos(t)} \mathcal{BV}(t,p)$. Moreover, we also define $t\upharpoonleft_p = \lambda\overline{x_k}.(t|_p)$, where $\overline{x_k} = seq_{bv}(t,p)$.

A *pattern* [14] is a term t for which all subterms $t|_p = X(\overline{t_n})$, with $X \in \mathcal{FV}(t)$ and $p \in MPos(t)$, satisfy the condition that $t_1{\downarrow_\eta}, \ldots, t_n{\downarrow_\eta}$ is a sequence of distinct elements of $\mathcal{BV}(t,p)$. Moreover, if all such subterms of t satisfy the additional condition $\mathcal{BV}(t,p) \setminus \{t_1{\downarrow_\eta}, \ldots, t_n{\downarrow_\eta}\} = \emptyset$, then the pattern t is *fully extended*. It is well known that unification of patterns is decidable and unitary [14]. Therefore, for every $t \in \mathcal{T}(\mathcal{F}_\perp, \mathcal{V})$ and pattern π, there exists at most one matcher between t and π, which we denote by $matcher(t, \pi)$. An *equation* is a multiset $\{\!\{s,t\}\!\}$, written $s == t$, where $s, t \in \mathcal{T}(\mathcal{F}_\perp, \mathcal{V})$ are terms of the same type.

In our theoretical framework, *programs* are considered as a special kind of conditional rewrite systems over fully extended linear patterns, with conditional equations between total terms.

Definition 1 (Programs). *A* Conditional Pattern Rewrite System *(*CPRS *for short) is a finite set of conditional rewrite rules of the form* $f(\overline{l_n}) \to r \Leftarrow C$*:*

- $f(\overline{l_n})$ *and* r *are total terms of the same base type,*
- $f(\overline{l_n})$ *is a fully extended linear pattern, and*
- C *is a (possibly empty) finite sequence of equations between total terms. In symbols,* $C \equiv \overline{s_m == t_m}$*, with* $s_i, t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ *for* $i = 1, \ldots, m$*.*

The term $f(\overline{l_n})$ *is called the left hand side* (lhs)*,* r *is the right hand side* (rhs)*, and* C *is the conditional part of the pattern rewrite rule.*

Each CPRS $\mathcal{R}$ induces a partition of $\mathcal{F}$ into $\mathcal{F}_d$ (*defined function symbols*) and $\mathcal{F}_c$ (*data constructors*):

$$\mathcal{F}_d = \{f \in \mathcal{F} \mid \exists (f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}\}, \quad \mathcal{F}_c = \mathcal{F} \setminus \mathcal{F}_d.$$

$\mathcal{R}$ is a *constructor-based* CPRS if each conditional pattern rewrite rule $f(\overline{l_n}) \to r \Leftarrow C$ satisfies the additional condition that $l_1, \ldots, l_n \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$.

4 The Higher-Order Rewriting Logic GHRC

In this section we extend the constructor-based Conditional ReWriting Logic CRWL from [5], in order to deal with conditional pattern rewrite rules. In contrast to *Meseguer's rewriting logic* and the *reflection* approach [13], which aims at modelling change caused by concurrent actions at a very high abstraction level, our rewriting logic intends to model the evaluation of λ-terms in a constructor-based language involving lazy functions. As in [5], we do not impose non-ambiguity conditions. This means that non-deterministic functions are allowed.

For all these reasons, we need to consider a (conditional) higher-order rewriting logic for declarative programming with non-strict and non-deterministic functions with call-time choice semantics, as an extension of the first-order rewriting logic CRWL. In order to obtain this aim, we propose this logic as the basis of a proof calculus, called GHRC (*Goal-oriented Higher-order Rewriting Calculus*), for reduction and joinability statements to a common value, designed as a generalization of the first-order proof system GORC which underlies the CRWL logic [5]. First, we need to define the suitable notion of *value* that is used in our setting with λ-abstractions and decidable higher-order unification.

Definition 2 (Values). *A* value *is a partial term t which has the following property: $\forall\, p \in MPos(t)$, $\forall\, (\pi \to r \Leftarrow C) \in \mathcal{R}$, $\nexists\, matcher(t\!\upharpoonright_p, \pi^{\updownarrow seq_{bv}(t,p)})$. In this definition, we implicitly assume that $\mathcal{FV}(t) \cap \mathcal{FV}(\pi) = \emptyset$. A* total value *is a value which is a total term. A* value substitution *is a substitution which binds variables to values. We write $Val(\mathcal{F}_\perp, \mathcal{V})$ (resp. $Val(\mathcal{F}, \mathcal{V})$) for the set of values (resp. total values), and $VSubst(\mathcal{F}_\perp, \mathcal{V})$ for the set of substitutions which bind variables to values.*

For a given CPRS $\mathcal{R}$ we want to derive statements of the following kind:

- *reduction statements*: $s \twoheadrightarrow t$, where $s, t \in \mathcal{T}(\mathcal{F}_\perp, \mathcal{V})$ are of the same type, whose intended meaning is that the term s can be reduced to t, so that the possibly partial term t approximates the denotation of s, as we will argue in Section 5.
- *equality statements*: $s == t$, which holds iff reduction statements $s \twoheadrightarrow u$ and $t \twoheadrightarrow u$ can be derived for some total value $u \in Val(\mathcal{F}, \mathcal{V})$.

The GHRC-provability relation is defined by the proof system given in Table 1. Note that GHRC-reduction is related to the idea of approximation, as shown by

Table 1. The GHRC proof calculus

B *Bottom*	$\lambda\overline{x_k}.\pi \twoheadrightarrow \lambda\overline{x_k}.\bot$
MN *Monotonicity*	$\dfrac{\lambda\overline{x_k}.s_1 \twoheadrightarrow \lambda\overline{x_k}.t_1 \ \cdots\ \lambda\overline{x_k}.s_n \twoheadrightarrow \lambda\overline{x_k}.t_n}{\lambda\overline{x_k}.a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.a(\overline{t_n})}$
RF *Reflexivity*	$s \twoheadrightarrow s$
OR *Outermost Reduction*	$\dfrac{\lambda\overline{x_k}.s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}}\theta \cdots \lambda\overline{x_k}.s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}}\theta \qquad \dfrac{C^{\updownarrow\overline{x_k}}\theta \qquad r^{\updownarrow\overline{x_k}}\theta \twoheadrightarrow u}{\boxed{\lambda\overline{x_k}.f(\overline{l_n^{\updownarrow\overline{x_k}}\theta}) \twoheadrightarrow u}}}{\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u}$ if $u \neq \lambda\overline{x_k}.\bot$, $\theta \in VSubst(\mathcal{F}_\bot, \mathcal{V})$, and $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$.
J *Join*	$\dfrac{s \twoheadrightarrow u \quad t \twoheadrightarrow u}{s == t}$ if $u \in Val(\mathcal{F}, \mathcal{V})$.

rule **B**. In rule **J**, we interpret equality (==) as joinability to a common total value u, since we wish to specify joinability as a generalization of strict equality, where total values in our higher-order framework play the same role as total constructor terms in the first-order framework (see [5]). Moreover, note that in rule **OR** for *Outermost Reduction* we use program rule instances $(f(\overline{l_n}) \to r \Leftarrow C)\theta$ with $\theta \in VSubst(\mathcal{F}_\bot, \mathcal{V})$ to reflect the so-called *call-time choice* for non-determinism (see the "coin example" in [5]). The other inference rules in GHRC are easier to understand.

Now, the main difference with respect to other similar proof systems is that the rule **OR** has been replaced by the consecutive application of two inference steps, **AR** for *Argument Reduction* and **FA** for *Function Application*, whose separate specification is displayed below:

$$\textbf{AR} \quad \frac{\lambda\overline{x_k}.s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}}\theta \quad \cdots \quad \lambda\overline{x_k}.s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}}\theta \qquad \lambda\overline{x_k}.f(\overline{l_n^{\updownarrow\overline{x_k}}\theta}) \twoheadrightarrow u}{\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u}$$

if $f \in \mathcal{F}_d$, $u \neq \lambda\overline{x_k}.\bot$, and $\theta \in VSubst(\mathcal{F}_\bot, \mathcal{V})$.

$$\textbf{FA} \quad \frac{C^{\updownarrow\overline{x_k}}\theta \qquad r^{\updownarrow\overline{x_k}}\theta \twoheadrightarrow u}{\lambda\overline{x_k}.f(\overline{l_n^{\updownarrow\overline{x_k}}\theta}) \twoheadrightarrow u} \quad \text{if } (f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R},\ \theta \in VSubst(\mathcal{F}_\bot, \mathcal{V}).$$

Taken together, these two rules say that a call to a function f is evaluated by computing approximated values for the arguments, and then applying a defining rule for f. The conclusion $\lambda\overline{x_k}.f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \twoheadrightarrow u$ introduces a so-called *basic fact*, which is only needed for debugging purposes in declarative programming, as we will argument in Section 6.

Detailed examples of GHRC-derivations in the form of *proof trees* in this kind of rewriting logics can be found in [5,18] and *Example 1* below. We write $\mathcal{R} \vdash \varphi$ if φ is a provable statement from a CPRS $\mathcal{R}$, $\mathcal{PT}(\varphi)$ for the set of proof trees for φ and $\mathcal{PT}_{\mathbf{L}}(\varphi)$ for the proof trees of $\mathcal{PT}(\varphi)$ which end with the application of an inference rule $\mathbf{L} \in \{\mathbf{B}, \mathbf{MN}, \mathbf{RF}, \mathbf{OR}, \mathbf{J}\}$. We also write $\mathcal{R} \vdash_{\mathbf{L}} \varphi$ if there exists a proof of $\mathcal{R} \vdash \varphi$ which ends with the application of rule $\mathbf{L}$, and $\mathcal{R} \nvdash_{\mathbf{L}} \varphi$ if there is no such a proof.

Finally, to complete the presentation of the higher-order rewriting logic GHRC in a declarative programming setting, we give a definition for the class of *goals* (from a given CPRS $\mathcal{R}$) and the set of *solutions* of a goal with which we are going to work.

Definition 3 (Goals and Solutions).

- *A* goal G *for a given* CPRS $\mathcal{R}$ *is a multiset* $\{\!\{\overline{s_n == t_n}\}\!\}$ *of equations between total terms of the same type. Equations are symmetric:* $s == t \equiv t == s$.
- $\gamma \in Subst(\mathcal{F}_\perp, \mathcal{V})$ *is a* solution *of a goal* $G \equiv \{\!\{\overline{s_n == t_n}\}\!\}$ *if* $\gamma\restriction_{\mathcal{FV}(G)} \in VSubst(\mathcal{F}_\perp, \mathcal{V})$*, and for each equation* $s_i == t_i$ *in* G *there exists a proof tree* $\mathcal{P}_i \in \mathcal{PT}(s_i\gamma == t_i\gamma)$*. The proof tree* $\mathcal{P}_i$ *is called a* witness *that* γ *is a solution of* $s_i == t_i$*. We write* $Soln(G)$ *for the set of solutions of a goal* G*.*

Example 1. For the CPRS of simple composition of circuits and hardware synthesis introduced in Section 1, we can check that the computed answer $\gamma = \{F \mapsto \lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v))\}$ is a solution of the goal $\{\!\{\, \lambda x, y.\, map\,(\lambda u, v.\, F(u,v), [(0,0),(x,1),(1,y)]) == [0,1,1],\ size\,(\lambda u, v.\, F(u,v)) == 3\,\}\!\}$. We have the following logical proof in the GHRC-calculus for $\mathcal{R} \vdash \lambda x, y.\, map\,(\lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v)), [(0,0),(x,1),(1,y)]) == [0,1,1]$, where $\mathcal{R}$ is the CPRS containing all the pattern rewrite rules mentioned in this example.

```
J $\lambda x, y.\, map\,(\lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v)), [(0,0),(x,1),(1,y)]) == [0,1,1]$
  OR $\lambda x, y.\, map\,(\lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v)), [(0,0),(x,1),(1,y)]) \to [0,1,1]$
      MN $\lambda x, y.\, [nand\,(nand\,(0,0), nand\,(0,0))\,|$
                  $map\,(\lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v)), [(x,1),(1,y)])] \to [0,1,1]$
        OR $nand\,(nand\,(0,0), nand\,(0,0)) \to 0$
            OR $nand\,(0,0) \to 1$
            OR $nand\,(1,1) \to 0$
        OR $\lambda x, y.\, map\,(\lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v)), [(x,1),(1,y)]) \to [1,1]$
            MN $\lambda x, y.\, [nand\,(nand\,(x,x), nand\,(1,1))\,|$
                        $map\,(\lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v)), [(1,y)])] \to [1,1]$
              OR $\lambda x.\, nand\,(nand\,(x,x), nand\,(1,1)) \to 1$
                    B $\lambda x.\, nand\,(x,x) \to \perp$
                  OR $nand\,(1,1) \to 0$
                  OR $nand\,(\perp,0) \to 1$
              OR $\lambda y.\, map\,(\lambda u, v.\, nand\,(nand\,(u,u), nand\,(v,v)), [(1,y)]) \to [1]$
```

$$\begin{array}{l}
\textbf{MN}\ \lambda y.\,[\mathit{nand}\,(\mathit{nand}\,(1,1),\mathit{nand}\,(y,y))\,|\\
\qquad\qquad \mathit{map}\,(\lambda u,v.\,\mathit{nand}\,(\mathit{nand}\,(u,u),\mathit{nand}\,(v,v)),[\,])\rightarrow[1]\\
\quad\ \textbf{OR}\ \lambda y.\,\mathit{nand}\,(\mathit{nand}\,(1,1),\mathit{nand}\,(y,y))\rightarrow[1]\\
\qquad\quad \textbf{OR}\ \mathit{nand}\,(1,1)\rightarrow 0\\
\qquad\qquad \textbf{B}\ \lambda y.\,\mathit{nand}\,(y,y)\rightarrow\bot\\
\qquad\quad \textbf{OR}\ \mathit{nand}\,(0,\bot)\rightarrow 1\\
\quad\ \textbf{OR}\ \mathit{map}\,(\lambda u,v.\,\mathit{nand}\,(\mathit{nand}\,(u,u),\mathit{nand}\,(v,v)),[\,])\rightarrow[\,]
\end{array}$$

The logical proof for $\mathcal{R}\vdash \mathit{size}\,(\lambda u,v.\,\mathit{nand}\,(\mathit{nand}\,(u,u),\mathit{nand}\,(v,v))) == 3$ is as follows:

$$\begin{array}{l}
\textbf{J}\ \mathit{size}\,(\lambda u,v.\,\mathit{nand}\,(\mathit{nand}\,(u,u),\mathit{nand}\,(v,v))) == 3\\
\quad \textbf{OR}\ \mathit{size}\,(\lambda u,v.\,\mathit{nand}\,(\mathit{nand}\,(u,u),\mathit{nand}\,(v,v))) \rightarrow 3\\
\qquad \textbf{OR}\ 1 + \mathit{size}\,(\lambda u,v.\,\mathit{nand}\,(u,u)) + \mathit{size}\,(\lambda u,v.\,\mathit{nand}\,(v,v)) \rightarrow 3\\
\qquad\quad \textbf{OR}\ \mathit{size}\,(\lambda u,v.\,\mathit{nand}\,(u,u)) \rightarrow 1\\
\qquad\qquad \textbf{OR}\ 1 + \mathit{size}\,(\lambda u,v.\,u) + \mathit{size}\,(\lambda u,v.\,u) \rightarrow 1\\
\qquad\qquad\quad \textbf{OR}\ \mathit{size}\,(\lambda u,v.\,u) \rightarrow 0\\
\qquad\quad \textbf{OR}\ \mathit{size}\,(\lambda u,v.\,\mathit{nand}\,(v,v)) \rightarrow 1\\
\qquad\qquad \textbf{OR}\ 1 + \mathit{size}\,(\lambda u,v.\,v) + \mathit{size}\,(\lambda u,v.\,v) \rightarrow 1\\
\qquad\qquad\quad \textbf{OR}\ \mathit{size}\,(\lambda u,v.\,v) \rightarrow 0
\end{array}$$

□

Finally, we give two results which characterizes the semantics proofs built with GHRC and generalizes useful known properties of CRWL-deductions for the first-order case (see [5,18] for more details and intuitive or informal explanations). The proof of Lemma 1 is given in [21].

Lemma 1 (Basic Semantic Property of GHRC-deductions). *Let* $s \in \mathit{Val}(\mathcal{F}_\bot,\mathcal{V})$. *If* $\mathcal{R}\vdash s \twoheadrightarrow t$ *then* $t\in \mathit{Val}(\mathcal{F}_\bot,\mathcal{V})$, $s \sqsupseteq t$, *and* $\mathcal{R}\nvdash_{\mathbf{OR}} s\twoheadrightarrow t$. *Moreover, if* $t\in \mathit{Val}(\mathcal{F},\mathcal{V})$ *then* $s\equiv t$.

Lemma 2. *If* $\mathcal{R}\vdash t\twoheadrightarrow u$, $p\in \mathit{Pos}(t)$ *is a* safe position *of* t *(i.e.,* $hd(t|_q) \in \mathcal{BV}(t,q)\cup\mathcal{F}_c$ *for each* $q\le p$*), and* $u\in \mathit{Val}(\mathcal{F},\mathcal{V})$, *then* $p\in \mathit{Pos}(u)$ *and* $\mathcal{R}\vdash t\rceil_p \twoheadrightarrow u\rceil_p$.

Proof. By induction on the length of p. By definition of GHRC, $\mathcal{R}\vdash_{\mathbf{L}} t \twoheadrightarrow u$, where $\mathbf{L}\in\{\mathbf{B},\mathbf{OR}\,(=\mathbf{AR}+\mathbf{FA}),\mathbf{RF},\mathbf{MN}\}$. We note that $\mathbf{L}\notin\{\mathbf{B},\mathbf{OR}\}$ because $p\in \mathit{Pos}(t)$ is a safe position and $u\in \mathit{Val}(\mathcal{F},\mathcal{V})$. If $\mathbf{L}=\mathbf{RF}$ then $t=u$ and Lemma 2 holds trivially. Otherwise $t=\lambda\overline{x_k}.a(\overline{t_n})$, $u=\lambda\overline{x_k}.a(\overline{u_n})$, and $\mathcal{R}\vdash\lambda\overline{x_k}.t_i\twoheadrightarrow\lambda\overline{x_k}.u_i$ for $i=1,\ldots,n$. If $p=1^k$ then Lemma 2 holds trivially. Otherwise $p=1^k.m.q$. Let $p'=1^k.q$, $t'=\lambda\overline{x_k}.t_m$, and $u'=\lambda\overline{x_k}.u_m$. Then p' is shorter than p, $\mathcal{R}\vdash t'\twoheadrightarrow u'$, $p'\in \mathit{Pos}(t')$ is a safe position, and $u'\in \mathit{Val}(\mathcal{F},\mathcal{V})$. By induction hypothesis for p' we obtain $p'\in \mathit{Pos}(u')$ a safe position, and $\mathcal{R}\vdash t'\rceil_{p'}\twoheadrightarrow u'\rceil_{p'}$. Since $t'\rceil_{p'}=t\rceil_p$ and $u'\rceil_{p'}=u\rceil_p$, we have $\mathcal{R}\vdash t\rceil_p\twoheadrightarrow u\rceil_p$. Also, $p'\in \mathit{Pos}(u')$ is a safe position, and $a\in\mathcal{F}_c\cup\{\overline{x_k}\}$ yields $p\in \mathit{Pos}(u)$ a safe position. □

5 Intended Models of CPRS-Programs

In this section, we briefly introduce some notions and results on the declarative semantics of CPRS-programs which are needed for the rest of the sections in

Table 2. The semantic calculus GHRC$_{\mathcal{I}}$

B$_{\mathcal{I}}$ *Bottom*	$\lambda\overline{x_k}.\pi \twoheadrightarrow \lambda\overline{x_k}.\bot$
MN$_{\mathcal{I}}$ *Monotonicity*	$\dfrac{\lambda\overline{x_k}.s_1 \twoheadrightarrow \lambda\overline{x_k}.t_1 \;\cdots\; \lambda\overline{x_k}.s_n \twoheadrightarrow \lambda\overline{x_k}.t_n}{\lambda\overline{x_k}.a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.a(\overline{t_n})}$
RF$_{\mathcal{I}}$ *Reflexivity*	$s \twoheadrightarrow s$
OR$_{\mathcal{I}}$ *Outermost Reduction*	$\dfrac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow t_1^{\updownarrow\overline{x_k}} \;\cdots\; \lambda\overline{x_k}.\, s_n \twoheadrightarrow t_n^{\updownarrow\overline{x_k}} \qquad u \twoheadrightarrow s}{\lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow s}$ if $u \neq \lambda\overline{x_k}.\,\bot$ and $(\lambda\overline{x_k}.\, f(\overline{t_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow u) \in \mathcal{I}$.
J$_{\mathcal{I}}$ *Join*	$\dfrac{s \twoheadrightarrow u \quad t \twoheadrightarrow u}{s == t}$ if $u \in Val(\mathcal{F}, \mathcal{V})$.

the paper. The semantic definition of *interpretation* is simpler than the one in the first-order setting [5,21] and our previous related work [21], where a more general notion of interpretation (under the name of *Algebra*) was presented. In our debugging scheme we will assume that the *intended model* of a CPRS-program is an interpretation.

Definition 4 (Interpretations and Models).

(1) A **basic fact** $\lambda\overline{x_k}.\, f(\overline{t_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow u$ *asserts that the (possibly non-linear) partial term* $u \in Val(\mathcal{F}_\bot, \mathcal{V})$ *approximates the result of* $f(\overline{t_n})$*, a fully extended linear pattern with the exact number of arguments expected by* f*'s arity, and with arguments* $t_i \in Val(\mathcal{F}_\bot, \mathcal{V})$*, which represent the partial approximations of* f*'s actual parameters needed to compute* u *as result. Moreover,* $f(\overline{t_n})$ *and* u *are partial terms of the same base type.*

(2) An **interpretation** $\mathcal{I}$ *is a set of basic facts fulfilling the following requirements for all* $f \in \mathcal{F}_d$ *with* $ar(f) = n$*, and* $f(\overline{t_n})$*,* $f(\overline{s_n})$ *fully extended linear patterns with* $\overline{t_n}, \overline{s_n} \in Val(\mathcal{F}_\bot, \mathcal{V})$ *arbitrary partial terms of the same base type that* $t, s \in Val(\mathcal{F}_\bot, \mathcal{V})$*:*

- $(\lambda\overline{x_k}.\, f(\overline{t_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow \lambda\overline{x_k}.\,\bot) \in \mathcal{I}$.
- *If* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow \lambda\overline{x_k}.\, t) \in \mathcal{I}$, $\lambda\overline{x_k}.\, t_i^{\updownarrow\overline{x_k}} \sqsubseteq \lambda\overline{x_k}.\, s_i^{\updownarrow\overline{x_k}}$, $\lambda\overline{x_k}.\, t \sqsupseteq \lambda\overline{x_k}.\, s$, *then also* $(\lambda\overline{x_k}.\, f(\overline{s_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow \lambda\overline{x_k}.\, s) \in \mathcal{I}$.

- *If* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\updownarrow \overline{x_k}}}) \twoheadrightarrow \lambda\overline{x_k}.\, t) \in \mathcal{I}$ *and* $\theta \in \mathit{VSubst}(\mathcal{F}_\perp, \mathcal{V})$*, then* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\updownarrow \overline{x_k}}\theta}) \twoheadrightarrow \lambda\overline{x_k}.\, t\theta) \in \mathcal{I}$*.*

A given reduction or equality statement φ *is valid in the interpretation* $\mathcal{I}$ *iff* φ *is a provable statement from* $\mathcal{I}$ *in the semantic calculus* $\mathit{GHRC}_\mathcal{I}$ *presented in Table 2. In general, for every basic fact* $\lambda\overline{x_k}.\, f(\overline{t_n^{\updownarrow \overline{x_k}}}) \twoheadrightarrow u$*, it can be proved that it is valid in* $\mathcal{I}$ *iff* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\updownarrow \overline{x_k}}}) \twoheadrightarrow u) \in \mathcal{I}$*. The denotation of a term* $t \in \mathcal{T}(\mathcal{F}_\perp, \mathcal{V})$ *is the set* $[\![\, t\,]\!]^\mathcal{I} = \{\, s \in \mathit{Val}(\mathcal{F}_\perp, \mathcal{V}) \mid t \twoheadrightarrow s \textit{ is valid in } \mathcal{I}\,\}$*.*

(3) $\mathcal{I}$ *is a* **model** *of a given CPRS* $\mathcal{R}$ *(i.e.,* $\mathcal{I} \models \mathcal{R}$*) iff every conditional pattern rewrite rule* $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$ *is valid in* $\mathcal{I}$ *(i.e.,* $\mathcal{I} \models f(\overline{l_n}) \to r \Leftarrow C$*): For any substitution* $\theta \in \mathit{VSubst}(\mathcal{F}_\perp, \mathcal{V})$ *and* $C \equiv \overline{s_m == t_m}$*, either* $[\![\, s_i\theta\,]\!]^\mathcal{I} \cap [\![\, t_i\theta\,]\!]^\mathcal{I} \cap \mathit{Val}(\mathcal{F}, \mathcal{V}) \neq \varnothing$ *(i.e.,* $\mathcal{I}$ *satisfies* $C\theta$*) and* $[\![\, f(\overline{l_n\theta})\,]\!]^\mathcal{I} \supseteq [\![\, r\theta\,]\!]^\mathcal{I}$*, or else* $\mathcal{I}$ *does not satisfy* $C\theta$*.*

Finally, from Definition 4 we can prove that the GHRC proof calculus is semantically sound. The proof is quite standard and details can be found in [21].

Theorem 1 (Semantic Correctness of GHRC). *If* $G \equiv \{\!\!\{\overline{s_n == t_n}\}\!\!\}$ *is a goal for a CPRS* $\mathcal{R}$ *and* $\gamma \in \mathit{Soln}(G)$ *then* $\gamma \in \mathit{Soln}_\mathcal{I}(G)$ *for all models* $\mathcal{I}$ *of* $\mathcal{R}$ *(i.e., every* $s_i\gamma == t_i\gamma$ *is valid in* $\mathcal{I}$*).*

6 Declarative Debugging of Wrong Answers in GHRC

In this section, we extend the declarative method for diagnosing wrong computed answers in first-order lazy functional-logic programs [1] to the higher-order setting of functional-logic programs with lambda abstractions.

Definition 5 (Symptoms and Errors). *Assume that* $\mathcal{I}$ *is the intended model for a given CPRS* $\mathcal{R}$*, and consider a substitution* $\gamma \in \mathit{VSubst}(\mathcal{F}, \mathcal{V})$ *produced as a computed answer for the goal* $G \equiv \{\!\!\{\overline{s_n == t_n}\}\!\!\}$ *by a goal solving system.*

(1) γ *is a* **wrong answer** *w.r.t.* $\mathcal{I}$ *(serving as a* **symptom***) iff* $\gamma \notin \mathit{Soln}_\mathcal{I}(G)$ *(i.e., there exists* $s_i == t_i$ *in* G *such that* $s_i\gamma == t_i\gamma$ *is not valid in* $\mathcal{I}$*).*

(2) $\mathcal{R}$ *is* **incorrect** *w.r.t.* $\mathcal{I}$ *iff there exists some conditional pattern rewrite rule* $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$ *(manifesting an* **error***) that is not valid in* $\mathcal{I}$ *(i.e.,* $\mathcal{I} \not\models f(\overline{l_n}) \to r \Leftarrow C$*).*

We say that a goal solving system is called *GHRC-sound* iff for any computed answer γ obtained for a goal G using a CPRS $\mathcal{R}$ we have that $\gamma \in \mathit{Soln}(G)$. The goal solving calculus $\mathrm{HOLN}^{\mathrm{DT}}$ given in [20] is GHRC-sound. This claim can be proved by a straightforward adaptation of the *soundness theorem* for $\mathrm{HOLN}^{\mathrm{DT}}$. Now we prove that the observation of an error symptom by any GHRC-sound goal solving system implies the existence of some error in the CPRS-program.

Theorem 2. *Assume that a GHRC-sound goal solving system computes* $\gamma \in \mathit{Subst}(\mathcal{F}, \mathcal{V})$ *as an answer for the goal* G *using a given CPRS* $\mathcal{R}$*. If* γ *is a wrong answer w.r.t. the user's intended model* $\mathcal{I}$ *then some conditional pattern rewrite rule belonging to* $\mathcal{R}$ *is not valid in* $\mathcal{I}$*.*

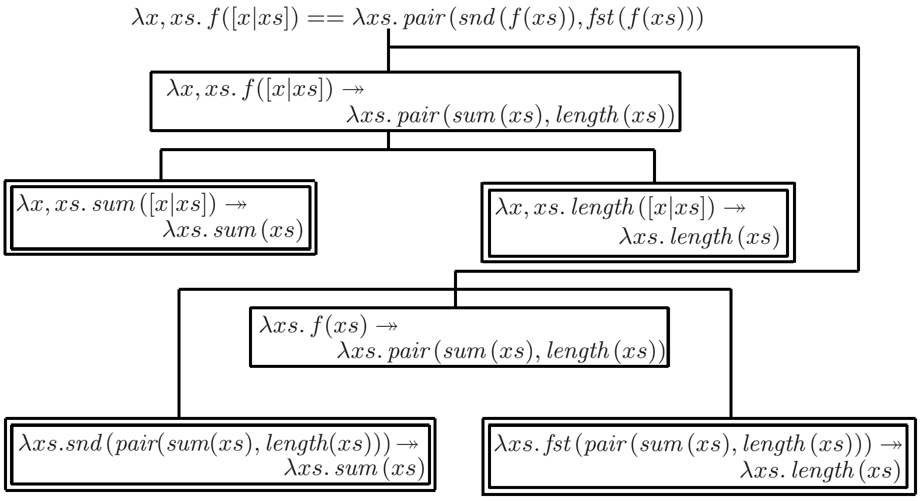

$\lambda x, xs.\, f([x|xs]) == \lambda xs.\, pair(snd(f(xs)), fst(f(xs)))$
$\lambda x, xs.\, f([x|xs]) \twoheadrightarrow \lambda xs.\, pair(sum(xs), length(xs))$
$\lambda x, xs.\, sum([x|xs]) \twoheadrightarrow \lambda xs.\, sum(xs)$
$\lambda x, xs.\, length([x|xs]) \twoheadrightarrow \lambda xs.\, length(xs)$
$\lambda xs.\, f(xs) \twoheadrightarrow \lambda xs.\, pair(sum(xs), length(xs))$
$\lambda xs. snd(pair(sum(xs), length(xs))) \twoheadrightarrow \lambda xs.\, sum(xs)$
$\lambda xs.\, fst(pair(sum(xs), length(xs))) \twoheadrightarrow \lambda xs.\, length(xs)$

Fig. 2. Computation tree in GHRC for declarative debugging with lambda abstractions

Proof. Because of the GHRC-soundness of the goal solving system, we know that $\gamma \in Soln(G)$. Then, from Theorem 1 we obtain $\gamma \in Soln_{\mathcal{J}}(G)$ for all model $\mathcal{J}$ of $\mathcal{R}$. Since γ is a wrong answer w.r.t. the user's intended model $\mathcal{I}$, it must be the case that $\gamma \notin Soln_{\mathcal{I}}(G)$ because of Definition 5. Therefore, we can conclude that the user's intended model $\mathcal{I}$ is not a model of $\mathcal{R}$. Then, by Definition 4, some conditional pattern rewrite rule belonging to $\mathcal{R}$ is not valid in $\mathcal{I}$. □

The debugging scheme proposed in [16] assumes that any terminated computation can be represented as a finite tree, called *computation tree*. The root of this tree corresponds to the result of the main computation, and each node corresponds to the result of some intermediate subcomputation. According to previous approaches in declarative debugging [1], our aim is to use *proof trees* in the GHRC proof calculus as computation trees. To this purpose, the only relevant nodes are those which correspond to the conclusion of **FA** steps. This is because all the other inference rules in GHRC, being program independent, cannot give rise to incorrect steps. The debugger works by navigating the computation tree, looking for erroneous nodes. Following the terminology of [16], an erroneous node with no erroneous children is called a *buggy node*.

The next theorem guarantees the logical correctness of declarative debugging with GHRC-proof trees for functional-logic programs with lambda abstractions:

Theorem 3 (Declarative Diagnosis of Wrong Answers). *Assume a wrong answer $\gamma \in Subst(\mathcal{F}, \mathcal{V})$, computed for the goal G using a given CPRS $\mathcal{R}$, such that $\gamma \notin Soln_{\mathcal{I}}(G)$, and $\mathcal{I}$ is the user's intended model of $\mathcal{R}$. Consider any GHRC-proof tree witnessing $\gamma \in Soln(G)$ as a computation tree, which must exist due to the existence of GHRC-sound goal solving systems. Then, declarative debugging has the following two properties:*

(a) **Completeness:** *navigating the computation tree will find a buggy node.*
(b) **Soundness:** *every buggy node in the computation tree points to a conditional pattern rewrite rule belonging to $\mathcal{R}$ which is not valid in $\mathcal{I}$.*

Proof. Item (a) follows immediately from the *Weak Completeness of Declarative Debugging* proved in [16], provided that the search strategy used to navigate the tree does not miss existing buggy nodes. To prove item (b), assume that the intended model is $\mathcal{I}$, and consider any given buggy node. This node must contain a basic fact $\lambda\overline{x_k}.\, f(\overline{l_n^{\updownarrow\overline{x_k}}\theta}) \twoheadrightarrow u$ which is not valid in $\mathcal{I}$ and has been inferred as the conclusion of a **FA** inference step using some conditional pattern rewrite rule belonging to $\mathcal{R}$, say $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$ and $\theta \in VSubst(\mathcal{F}_\perp, \mathcal{V})$. Therefore, the children of $\lambda\overline{x_k}.\, f(\overline{l_n^{\updownarrow\overline{x_k}}\theta}) \twoheadrightarrow u$ in the GHRC-proof tree, $C^{\updownarrow\overline{x_k}}\theta$ and $r^{\updownarrow\overline{x_k}}\theta \twoheadrightarrow u$ are valid in $\mathcal{I}$, because they are the children of a buggy node. With this we can conclude that $C^{\updownarrow\overline{x_k}}\theta$ and $r^{\updownarrow\overline{x_k}}\theta \twoheadrightarrow u$ are valid in $\mathcal{I}$ (i.e., $\mathcal{I}$ satisfies $C^{\updownarrow\overline{x_k}}\theta$ and $u \in [\![r^{\updownarrow\overline{x_k}}\theta]\!]^{\mathcal{I}}$), while $\lambda\overline{x_k}.\, f(\overline{l_n^{\updownarrow\overline{x_k}}\theta}) \twoheadrightarrow u$ is not valid in $\mathcal{I}$ (i.e., $u \notin [\![\lambda\overline{x_k}.\, f(\overline{l_n^{\updownarrow\overline{x_k}}\theta})]\!]^{\mathcal{I}}$). Then $[\![r^{\updownarrow\overline{x_k}}\theta]\!]^{\mathcal{I}} \nsubseteq [\![\lambda\overline{x_k}.\, f(\overline{l_n^{\updownarrow\overline{x_k}}\theta})]\!]^{\mathcal{I}}$, which means (see Definition 4) that the conditional pattern rewrite rule $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$ is not valid in $\mathcal{I}$. □

Example 2. For the particular function $f \to \lambda xs.\ pair(sum(xs),\ length(xs))$, where *pair* is a data constructor, and

$$\begin{array}{llllll}
sum\,([\,]) & \to 0 & length\,([\,]) & \to 0 & fst\,(pair\,(x,y)) & \to x \\
sum\,([x|xs]) & \to x{+}sum(xs) & length\,([x|xs]) & \to 1{+}length(xs) & snd\,(pair(x,y)) & \to y
\end{array}$$

we can check that $\gamma = \{E \mapsto pair(0,0), G \mapsto \lambda u, z.pair(u + fst(z), 1 + snd(z))\}$ is a solution of the goal $\{\!\{ f([\,]) == E,\ \lambda x, xs.\, f([x|xs]) == \lambda x, xs.\, G(x,\ f(xs)) \}\!\}$. Now we suppose that we obtain the solution $\{G \mapsto \lambda z.\, pair(snd(z), fst(z))\}$. We know that this is a *wrong computed answer*, but we don't know exactly why. For this reason, we decide to explore the corresponding *computation tree* (see Fig. 2). Looking at the leaves of this tree, we find two *buggy nodes* (represented by double rectangles) concerning to the application of the functions *fst* and *snd*, respectively. We note that we have written erroneous pattern rewrite rules: $fst\,(pair\,(x,y)) \to y$ and $snd\,(pair\,(x,y)) \to x$. Moreover, we also learn that the application of the functions *sum* and *length* is also erroneous, because we have another two *buggy nodes*, suggesting that we have written again two incorrect pattern rewrite rules: $sum\,([x|xs]) \to sum\,(xs)$ and $length\,([x|xs]) \to length\,(xs)$. We can correct all of them to obtain the CPRS represented above, and the right computed answer for the goal if we repeat the computation. □

7 Conclusions, Related and Future Work

We have presented a generalization of the well-known Naish's declarative method for diagnosing wrong computed answers in first-order lazy functional-logic programming [1] to the more expressive setting of the simply typed λ-calculus with decidable higher-order unification [14], where the notion of lazy and possibly non-deterministic higher-order function [5] plays a central role (in comparison with related work [4,7]). Moreover, we have used the powerful and generic semantic framework introduced in our previous work [21] to define a more suitable and specific model-theoretic semantics for declarative debugging based only on the simple notion of interpretation and the intended model of a CPRS-program.

All of the many higher-order extensions of functional-logic languages are, to our knowledge, limited to first-order unification and are not complete in our higher-order sense. For instance, the work [10] uses higher-order variables, but only (first-order) narrowing on first-order terms plus β-reduction as the operational model. The work [4] similarly permits higher-order variables. Higher-order unification is currently used in theorem provers like *Isabelle* [12], and for higher-order logic programming in the language *λProlog* [15]. Other currently developed functional-logic languages such as *Oz* [17], *Escher* [9] and *Curry* [6] do not utilize higher-order unification and hence do not focus on a theoretical framework for declarative debugging as the approach presented in this paper.

Planned future work will include further theoretical investigation to integrate non-equality constraints (e.g., see $-\pi/4 \leq f(x) \leq \pi/2$ and $f(x) \neq 0$ in Section 1) in the conditional part of pattern rewrite rules and goals, following the line of recent researches on *constraint rewriting logics* [3,19]. On the practical side, we are currently working on the development of a debugger which implements our declarative debugging method, following the line of previous works [1,2]. A difficult issue in the construction of reasonable higher-order functional-logic programming languages is the high degree of non-determinism that must be explored to compute all the answers. Without higher-order patterns, this is managed by sophisticated demand-driven strategies to restrict the degree of non-determinism (see [18]). Adding higher-order features increases the search space dramatically. For this purpose, we have presented in [20] an appropriate operational semantics based on narrowing with higher-order overlapping *definitional trees*, a useful tool for achieving a demand-driven strategy which avoids unneeded steps to reduce the huge search space for bindings of higher-order variables. Although in this paper we propose only a more specific language to construct proofs for debugging and verification purposes, we believe that the integration of appropriate *definitional proof trees* could be useful to the development of efficient debuggers in our higher-order logical framework.

Acknowledgments. The authors are thankful to Mario Rodríguez Artalejo and Mircea Marin for their collaboration, comments and contributions during the first stages of the development of this work.

References

1. Caballero, R., López-Fraguas, F.J., Rodríguez-Artalejo, M.: Theoretical Foundations for the Declarative Debugging of Lazy Functional-Logic Programs. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 170–184. Springer, Heidelberg (2001)

2. Caballero, R., Rodríguez-Artalejo, M.: $\mathcal{DDT}$: A Declarative Debugging Tool for Functional-Logic Languages. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 70–84. Springer, Heidelberg (2004)
3. López-Fraguas, F.J., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: A New Generic Scheme for Functional-Logic Programming with Constraints. Journal of Higher-Order and Symbolic Computation 20(1/2), 73–122 (2007)
4. González-Moreno, J.C., Hortalá-González, M.T., Rodríguez-Artalejo, M.: A higher-order rewriting logic for functional-logic programming. In: Proc. ICLP 1997, pp. 153–167 (1997)
5. 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)
6. Hanus, M.: Curry: an Integrated Functional Logic Language., `http://www-i2.informatik.uni-kiel.de/~curry/`
7. Hanus, M., Prehofer, C.: Higher-order narrowing with definitional trees. Journal of Functional Programming 9(1), 33–75 (1999)
8. Hindley, J.R., Seldin, J.P.: Introduction to Combinatorics and λ-Calculus. Cambridge University Press, Cambridge (1986)
9. Lloyd, J.W.: Combining functional and logic programming languages. In: Proc. ILPS 1994 (1994)
10. Lock, H.C.: The Implementation of Functional-Logic Languages. Oldenbourg Verlag (1993)
11. López-Fraguas, F.J., Sánchez-Hernández, J.: $\mathcal{TOY}$: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999), `http://toy.sourceforge.net`
12. Paulson, L.C.: Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
13. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. TCS 96, 73–155 (1992)
14. Miller, D.: A logic programming language with λ-abstraction, function variables, and simple unification. Journal of Logic and Computation 1(4), 497–536 (1991)
15. Nakahara, K., Middeldorp, A., Ida, T.: A complete narrowing calculus for higher-order functional-logic programming. In: Swierstra, S.D. (ed.) PLILP 1995. LNCS, vol. 982, pp. 97–114. Springer, Heidelberg (1995)
16. Naish, L.: A Declarative Debugging Scheme. Journal of Functional and Logic Programming (1997)
17. Smolka, G.: The Definition of Kernel Oz. Technical Report DFKI Research Report RR-94-23, Saarbrücken, Germany (1994)
18. del Vado-Vírseda, R.: A Demand-driven Narrowing Calculus with Overlapping Definitional Trees. In: Proc. PPDP 2003, pp. 253–263. ACM, New York (2003)
19. del Vado Vírseda, R.: Declarative Constraint Programming with Definitional Trees. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 184–199. Springer, Heidelberg (2005)
20. del Vado Vírseda, R.: A Higher-Order Demand-Driven Narrowing Calculus with Definitional Trees. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 169–184. Springer, Heidelberg (2007)
21. del Vado-Vírseda, R.: A Higher-Order Logical Framework for the Algorithmic Debugging and Verification of Declarative Programs. In: Proc. PPDP 2009, pp. 49–60. ACM Press, New York (2009)

Author Index