Santiago Escobar (Ed.)

Functional and (Constraint) Logic Programming

18th International Workshop, WFLP'09

part of the Federated Conference on Rewriting, Deduction, and Programming (RDP'09)

Brasília, Brazil, June 28, 2009.

Informal Proceedings

Preface

This report contains the informal workshop proceedings of the *18th International Workshop on Functional and (Constraint) Logic Programming* (WFLP'09), held at Brasília, Brazil, during June 28, 2009. WFLP'09 is part of the Federated Conference on Rewriting, Deduction, and Programming (RDP'09). Previous meetings are: WFLP 2008 (Siena, Italy), WFLP 2007 (Paris, France), WFLP 2006 (Madrid, Spain), WCFLP 2005 (Tallinn, Estonia), WFLP 2004 (Aachen, Germany), WFLP 2003 (Valencia, Spain), WFLP 2002 (Grado, Italy), WFLP 2001 (Kiel, Germany), WFLP 2000 (Benicassim, Spain), WFLP'99 (Grenoble, France), WFLP'98 (Bad Honnef, Germany), WFLP'97 (Schwarzenberg, Germany), WFLP'96 (Marburg, Germany), WFLP'95 (Schwarzenberg, Germany), WFLP'94 (Schwarzenberg, Germany), WFLP'93 (Rattenberg, Germany), and WFLP'92 (Karlsruhe, Germany).

The aim of the WFLP workshop 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.

The Program Committee of WFLP'09 collected three reviews for each paper and held an electronic discussion during May 2009. The Program Committee selected 12 regular papers for presentation at the workshop. In addition to the selected papers, the scientific program includes two invited lectures 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.

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. We gratefully acknowledge the *Departamento de Sistemas Informáticos y Computación* of the *Universidad Politécnica de Valencia*, who has supported this event. Finally, we express our gratitude to all members of the local organization of the Federated Conference on Rewriting, Deduction, and Programming (RDP'09), whose work has made the workshop possible.

Brasília, Brazil, Santiago Escobar
June 2009 WFLP'09 Chair

Organization

WFLP'09 is part of the Federated Conference on Rewriting, Deduction, and Programming (RDP'09).

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-Universitat Munster, 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 Referees

Gloria Álvarez	Iliano Cervesato	Albert Rubio	Rafael del Vado Vírseda
Demis Ballis	Yukiyoshi Kameyama	Clara Segura	Toshiyuki Yamada
Bernd Braßel	Temur Kutsia	Peter Sestoft	Hans Zantema
Linda Brodo	Miguel Palomino	Thierry Boy de la Tour	

Sponsoring Institution

Departamento de Sistemas Informáticos y Computación (DSIC)
Universidad Politécnica de Valencia (UPV)

Table of Contents

Strategic Deduction

Claude Kirchner, Florent Kirchner, and Hélène Kirchner

INRIA
France

Abstract. In previous works, we have introduced the notion of abstract strategies for abstract reduction systems and defined adequate properties of termination, confluence and normalization under strategies. Thanks to this abstract strategy concept, we draw a parallel between strategies for computation and strategies for deduction. Then, deduction rules can be viewed as rewrite rules, a deduction step as a rewriting step and a proof construction step as a narrowing step for an adequate abstract reduction system, possibly in constraint handling settings.

The fundamental complementarity between deduction and computation, as emphasized in particular in deduction modulo [9], gives now rise to a completely new generation of proof assistants where customized deductions are performed modulo appropriate and user definable computations [5,6]. This has in particular the advantage to allow for a uniform implementation of higher-order and first-order logics [8,7] making possible the safe use of existing dedicated proof environments [16,10,4]. This generalizes classical approaches used in first-order theorem proving [17], as well as higher-order ones like PVS [18], TPS [1,2], Omega [3,19], Coq [11] or Mizar [20], to mention just a few.

Proof search in these environments goes back to the late sixties and then throught the design of ML as the metalanguage of LCF. It requires to guide proof discovery using so called strategies, tactics, tacticals or proof plans, terms widely used in artificial intelligence, in automated or interactive reasoning, in semantics of programming languages—as well as in every day life.

The collusion of deduction and computation in next-generation proof assistants has inspired our recent attempt at providing an uniform (domain-agnostic, if you will) definition for strategies, starting from a rule-based view point [13].

For term rewriting, reduction strategies study which expressions should be selected for evaluation and which rules should be applied. These choices are usualy made to increase the efficiency of evaluation but may affect fundamental properties of computations such as confluence or (non-)termination. Programming languages like TOM[1], ELAN, Maude and Stratego allow for the explicit definition of the evaluation strategy, whereas languages like Clean, Curry, and Haskell allow for its modification.

In theorem proving environments, including automated theorem provers, proof checkers, and logical frameworks, strategies (also called tacticals in some

[1] `http://tom.loria.fr`

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.

In this talk, we will recall the theoretical foundations of strategies and the convergence of two points of view, namely rewriting-based computations on one hand, rule-based deduction and proof-search on the other hand.

While strategies for computation [12] essentially rely on the largely explored and well-known domain of term reduction by rewriting or narrowing, strategies for deduction require to introduce an original point of view: we define deduction rules as rewrite rules, a deduction step as a rewriting step, a deduction system as an abstract reduction system. Proof construction in this context becomes narrowing derivation. Computation, deduction and proof search are then captured by the foundational concept of abstract strategy.

Time permitting, we will show how deduction and proof search under constraints could be investigated this way, especially using antipatterns [15,14].

References

1. Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A Theorem Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, June 1996.
2. Peter B. Andrews and Chad E. Brown. TPS: A hybrid automatic-interactive system for developing proofs. *J. Applied Logic*, 4(4):367–395, 2006.
3. Christoph Benzmüller, Matthew Bishop, and Volker Sorge. Integrating TPS and Omega. *J. UCS*, 5(3):188–207, 1999.
4. Frédéric Blanqui, Jean-Pierre Jouannaud, and Pierre-Yves Strub. Building Decision Procedures in the Calculus of Inductive Constructions. In Jacques Duparc and Thomas Henziger, editors, *16th Annual Conference on Computer Science and Logic - CSL 2007*, volume 4646 of *Lecture Notes in Computer Science*, Lausanne, Suisse, 2007. Springer Verlag.
5. Paul Brauner, Clément Houtmann, and Claude Kirchner. Principle of superdeduction. In Luke Ong, editor, *Proceedings of LICS*, pages 41–50, jul 2007.
6. Paul Brauner, Clément Houtmann, and Claude Kirchner. Superdeduction at work. In Hubert Comon, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof. Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600. Springer, jun 2007.
7. Guillaume Burel. Superdeduction as a Logical Framework. submitted, jan 2008.
8. Denis Cousineau and Gilles Dowek. Embedding Pure Type Systems in the lambda-Pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer-Verlag, 2007.
9. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003.
10. Gilles Dowek and Benjamin Werner. Arithmetic as a Theory Modulo. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 423–437. Springer-Verlag, 2005.
11. Bruno Barras et al. *The Coq Proof Assistant Reference Manual*, 2006.

12. Claude Kirchner. Strategic Rewriting. *Electr. Notes Theor. Comput. Sci. Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming - WRS'2004, Aachen, Germany*, 124(2):3–9, 2005.
13. Claude Kirchner, Florent Kirchner, and Hélène Kirchner. Strategic Computations and Deductions. In *Reasoning in Simple Type Theory*, volume 17 of *Mathematical Logic and Foundations*. College Publications, 2008.
14. Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-Pattern Matching. In *16th European Symposium on Programming (ESOP'07)*, Braga, Portugal, 2007.
15. Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-Pattern Matching Modulo. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers*, volume 5196 of *Lecture Notes in Computer Science*, pages 275–286, Tarragona, Spain, 2008. Springer.
16. Florent Kirchner and Claudio Sacerdoti Coen. The Fellowship proof manager. `www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship/`, 2007.
17. William McCune. Semantic Guidance for Saturation Provers. *Artificial Intelligence and Symbolic Computation*, pages 18–24, 2006.
18. Sam Owre, John Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.
19. Jörg H. Siekmann, Christoph Benzmüller, and Serge Autexier. Computer supported mathematics with Omega. *J. Applied Logic*, 4(4):533–559, 2006.
20. A. Trybulec and H. Blair. Computer Aided Reasoning with Mizar. In R. Parikh, editor, *Logic of Programs*. Springer Verlag, New York, 1985.

Programming with Multiple Paradigms in Lua

Roberto Ierusalimschy

PUC-Rio

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 by 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].

Two key points in the design of the language that led to those uses 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, and reflexive capabilities [3, 4]. So, although Lua is primarily a procedural language, it can be, and frequently is, used in several different programming paradigms, such as functional, object-oriented, goal-oriented, and concurrent programming, and also for data description.

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

2 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 this:

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

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

However, this syntax is simply sugar for an assignment:

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

(This is quite similar to a `define` in Scheme [5].)

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 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 [6].

Most implementations of first-class functions with lexical scoping neglect assignment. Pure functional languages do not have assignment. In ML assignable cells have no names, so the problem does not arise. Some Scheme compilers (e.g., Orbit [7]) actually implement assignable variables as ML cells (*assignment conversions*), on the correct ground that they are not used often.

None of those implementations fit Lua, a procedural language where assignment is the norm. Lua has added requirements 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 where a nested function could access the value of an outer variable, but could not assign to such variable. Lua version 5, released in 2003, came with a novel technique for implementing closures that satisfies the following requirements [8]:

- 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.

3 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.

One missing ingredient 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 new 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 [9, 10]. 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 as before. Otherwise, Lua looks for that key in the parent of `obj`. (If the parent object has a parent, this query may trigger another query in the parent's parent and so on.) Once it found a value for that key, Lua calls it with the original object `obj` as the first argument, so that `obj` becomes the value of the parameter `self` inside the method's body.

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).

4 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 pattern matching [11] and Prolog-like queries [12].

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, we may implement a problem solver using a backtracking mechanism that successively tries each alternative until it finds an adequate result.

Although Lua does not offer any specific mechanism for this kind of problem solving, we can use Lua coroutines [13] for the task. A well-known model for Prolog-style backtracking is the *two-continuation model* [14, 15], which needs

```
-- 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. 1. A simple pattern-matching library.

multi-shot continuations. However, it is not difficult to adapt the model to coroutines that are equivalent to one-shot continuations [16, 17]. 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 1 shows a simple implementation of a pattern-matching library, taken from [17]. Each pattern is represented by a function that receives the subject plus the current position and yields different final positions.

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 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).

5 Concurrent Programming

Lua avoids the problems of imperative programming with multithreading by cutting either preemption or shared memory.

Lua uses coroutines to achieve multithreading without preemption. A *stackful* coroutine [17] is essentially a thread; it is easy to write a simple scheduler with a few lines of code to complete the system [3]. 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 difficulties. 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.

Lua may also achieve multithreading by cutting shared memory. In this case, a program creates several independent Lua states that behave like Unix processes. Each state has its own logical memory space with independent garbage collection. All communication is done through message passing. Messages may contain only primitive values, such as numbers or strings, because references (addresses) have no meaning across different states. A main advantage of multiple states 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 already offers multiple states: 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. However, the creation of new states can be done only in C and the

[2] Simula offered coroutines for this reason [18].

```
dan = name{first = "Daniel", last = "Friedman"}
mitch = name{last = "Wand",
             first = "Mitchell",
             middle = "P."}
chris = name{first = "Christopher", last = "Haynes"}
book{
  author = {dan, mitch, chris},
  title = "Essentials of Programming Languages",
  edition = 2,
  year = 2001,
  publisher = "The MIT Press"
}
```

Fig. 2. Data description with SOL/Lua.

communication between states also needs some C code. So, multi-state concurrency cannot be implemented in pure Lua; it needs some external support written in C. Currently there are at least two libraries with such support: LuaLanes [19], which uses tuple spaces for communication, and Luaproc [20], which uses mailboxes.

6 Data Description

Lua was born from a data-description language, called SOL [21], 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 in BibTeX, a tool for creating and formating lists of bibliographic references. A main difference between SOL and BibTeX was that SOL had the ability to declare and nest declarations. Figure 2 shows a typical fragment, 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 2 is an imperative program. The syntax `{first = "Daniel", ...}` is a *constructor*: it builds a table, 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 [22]. GUPPY uses Lua to describe sequence annotation data from genome databases [23]. Some descriptions comprise thousands of elements running for a few million lines of 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.

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 [24] 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 (2006)
2. Associação Brasileira de Normas Técnicas: Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiodifusão digital. (2007) ABNT NBR 15606-2.
3. Ierusalimschy, R.: Programming in Lua. second edn. Lua.org, Rio de Janeiro, Brazil (2006)
4. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: Lua 5.1 Reference Manual. Lua.org, Rio de Janeiro, Brazil (2006)
5. Kelsey, R., Clinger, W., Rees, J.: Revised^5 report on the algorithmic language Scheme. Higher-Order and Symbolic Computation **11**(1) (August 1998) 7–105
6. 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. Lua.org (2008) 97–107
7. 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'86).
8. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: The implementation of Lua 5.0. In: IX Brazilian Symposium on Programming Languages, Recife, PE (May 2005) 63–75
9. Ungar, D., Smith, R.B.: Self: The power of simplicity. SIGPLAN Notices **22**(12) (December 1987) 227–242 (OOPLSA'87).

10. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. SIGPLAN Notices **21**(11) (November 1986) 214–223 (OOPLSA'86).
11. Griswold, R., Griswold, M.: The Icon Programming Language. Prentice-Hall, New Jersey, NJ (1983)
12. Clocksin, W., Mellish, C.: Programming in Prolog. Springer-Verlag (1981)
13. de Moura, A.L., Rodriguez, N., Ierusalimschy, R.: Coroutines in Lua. Journal of Universal Computer Science **10**(7) (July 2004) 910–925 (SBLP 2004).
14. Haynes, C.T.: Logic continuations. J. Logic Programming **4** (1987) 157–176
15. Wand, M., Vaillancourt, D.: Relating models of backtracking. In: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, Snowbird, UT, ACM (September 2004) 54–65
16. Ierusalimschy, R., de Moura, A.L.: Some proofs about coroutines. Monografias em Ciência da Computação 04/08, PUC-Rio, Rio de Janeiro, Brazil (2008)
17. de Moura, A.L., Ierusalimschy, R.: Revisiting coroutines. ACM Transactions on Programming Languages and Systems **31**(2) (2009) 6.1–6.31
18. Birtwistle, G., Dahl, O., Myhrhaug, B., Nygaard, K.: Simula Begin. Petrocelli Charter (1975)
19. Kauppi, A.: Lua Lanes — multithreading in Lua. (2009) `http://kotisivu.dnainternet.net/askok/bin/lanes/`.
20. Skyrme, A., Rodriguez, N., Ierusalimschy, R.: Exploring Lua for concurrent programming. In: XII Brazilian Symposium on Programming Languages, Fortaleza, CE (August 2008) 117–128
21. 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) 2.1–2.26
22. Koyama, T., et al.: Simulation tools for damping in high frequency resonators. In: 4th IEEE Conference on Sensors, IEEE (October 2005) 349–352
23. Ueno, Y., Arita, M., Kumagai, T., Asai, K.: Processing sequence annotation data using the Lua programming language. Genome Informatics **14** (2003) 154–163
24. Fernandes, F., Batista, T.: Dynamic aspect-oriented programming: An interpreted approach. In: Proceedings of the 2004 Dynamic Aspects Workshop (DAW04). (March 2004) 44–50

A Theoretical Framework for the Declarative Debugging of Functional Logic Programs with Lambda Abstractions *

Ignacio Castiñeiras Pérez and Rafael del Vado Vírseda

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
{ncasti,rdelvado}@sip.ucm.es

Abstract. In this paper, we extend the declarative method 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 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 and Motivation

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 [3, 4] 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 *loves*, given by first-order conditional rewrite rules ($\rightarrow$) where the conditional part (formed only by equations $==$) is delimited by the "$\Leftarrow$" symbol:

* This work has been partially supported by the Spanish National Projects TIN2008-06622-C03-01, TIN2005-09027-C03-03, S-0505/TIC/0407, and UCM-BSCH-GR58/08-910502.

$$
\begin{array}{lllll}
loves\,(john, mary) & \rightarrow & true & & \\
loves\,(mary, Y) & \rightarrow & true & \Leftarrow & likes\,(Y, wine) \quad == false \\
loves\,(X, mary) & \rightarrow & true & \Leftarrow & loves\,(mary, X) == true
\end{array}
$$

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 ($\rightarrow$) statements. The most important result is the existence of sound and complete *lazy narrowing calculi* [3, 4, 11] for solving goals in first-order CRWL-theories presented by CTRS-programs. Moreover, a higher-order extension of CRWL is presented in [2] 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) 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, we consider the following conditional pattern rewrite system (adapted from [5]) defining a higher-order function *diff*, where $diff\,(f, x)$ computes the differential of a function f at some point x:

$$
\begin{array}{lllll}
diff\ (\lambda y.\, y, x) & \rightarrow & 1 & & \\
diff\ (\lambda y.\, sin(f(y)), x) & \rightarrow & cos(f(x)) * diff\ (\lambda y.\, f(y), x) & \Leftarrow & -\pi/2 \leq f(x) \leq \pi/2 \\
diff\ (\lambda y.\, ln(f(y)), x) & \rightarrow & diff\,(\lambda y.\, f(y), x)/f(x) & \Leftarrow & f(x) \neq 0
\end{array}
$$

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* [5] 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 [3, 4, 11, 13].

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 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 in [10]. 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

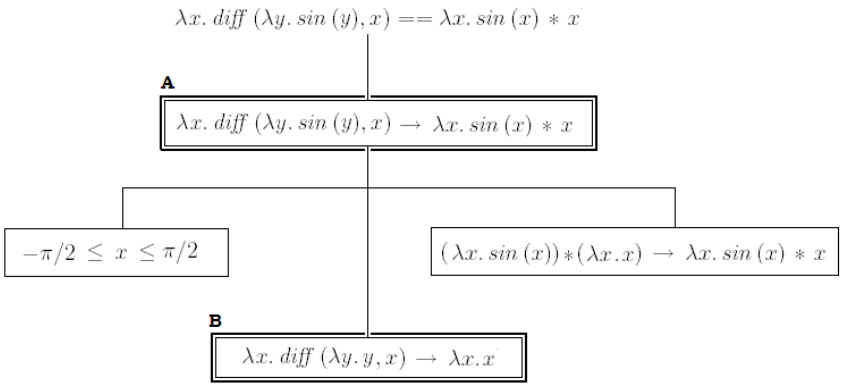


Fig. 1. Computation tree for declarative debugging involving lambda abstractions

example. The following higher-order functional logic program involving lambda abstractions is an erroneous fragment of the previous conditional pattern rewrite system to compute the differential of a function:

$$\begin{array}{lll} diff\ (\lambda y.\ y, x) & \rightarrow & \underline{x} \\ diff\ (\lambda y.\ sin(f(y)), x) & \rightarrow & \underline{sin}(f(x)) * diff\ (\lambda y.\ f(y), x) \Leftarrow -\pi/2 \leq f(x) \leq \pi/2 \end{array}$$

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., [5, 13]). For instance, we consider a goal to compute appropriate functions $\lambda x, y.\ F(x, y)$ and $\lambda x.\ D(x)$ for which the differential of $\lambda y.\ sin\ (F(x, y))$ at some point $x \in [-\pi/2, \pi/2]$ satisfies $\lambda x.\ diff\ (\lambda y.\ sin\ (F(x, y)), x) == \lambda x.\ D(x)$. We obtain a substitution to represent the solution $\{F \mapsto \lambda x, y.\ y, D \mapsto \lambda x.\ sin\ (x) * x\}$ under the constraint $-\pi/2 \leq x \leq \pi/2$. The user regards this solution as incorrect (because the user really expects as solution $\{F \mapsto \lambda x, y.\ y, D \mapsto \lambda x.\ cos\ (x)\}$) according to their own *intended model* 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 [5, 13]. The computation tree for our current example is graphically represented in **Fig. 1** (more on its structure and construction will be explained in Section 5). 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 model $\mathcal{I}$.

For instance, the computation tree depicted in **Fig. 1** has a buggy node in *node B* because $\lambda x.\, x$ is not the differential of the function $\lambda x.\, \mathit{diff}(\lambda y.\, y, x)$. Therefore, the first pattern rewrite rule $\mathit{diff}\ (\lambda y.\, y, x) \to x$ is incorrect w.r.t. the user's intended program semantics; as we have shown previously, the first rule should be $\mathit{diff}\ (\lambda y.\, y, x) \to 1$. After this correction, there is another buggy node in *node A* because the differential of $\lambda y.\, \mathit{sin}\,(y)$ at point x is $\mathit{cos}\,(x)$ instead of $\mathit{sin}\,(x)$. Indeed, the second conditional pattern rewrite rule is also incorrect w.r.t. the user's intended model of the program as we have previously seen. After this new correction, no more wrong computed answers will be observed for the goal discussed above, and the right solution $\{F \mapsto \lambda x, y.\, y, D \mapsto \lambda x.\, \mathit{cos}\,(x)\}$ is then obtained.

The paper is structured as follows. In Section 2 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 3 we introduce a 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 4 is concerned with the model-theoretic semantics for GHRC-programs. In Section 5 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 6 summarizes some conclusions and presents a brief outline of planned future work.

2 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., [5]). 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 "$\to$". 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 $t\downarrow_\beta^\eta$, is the η-expanded form of the β-normal form of t. It is well-known that $s =_{\alpha\beta\eta} t$ if $s\downarrow_\beta^\eta =_\alpha t\downarrow_\beta^\eta$ [6]. 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 $\mathit{hd}(t)$. 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:

$$\lambda\overline{x_k}.\perp \sqsubseteq \lambda\overline{x_k}.t \qquad t \sqsubseteq t \qquad \frac{s_1 \sqsubseteq t_1 \ \cdots \ s_n \sqsubseteq t_n}{\lambda\overline{x_k}.a(\overline{s_n}) \sqsubseteq \lambda\overline{x_k}.a(\overline{t_n})}$$

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:* $\mathit{Pos}(\lambda\overline{x_k}.a(\overline{t_n})) = \{1^i \mid 0 \le i \le k\} \cup \{1^k.j.q \mid 1 \le j \le n,\ q \in \mathit{Pos}(t_j)\}$, 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 $p \in \mathit{Pos}(t)$:*

$$(\lambda\overline{x_k}.a(\overline{t_n}))|_p = \begin{cases} \lambda x_{i+1} \ldots x_k.a(\overline{t_n}) & \text{if } p = 1^i \text{ with } 0 \le i \le k, \\ t_i|_q & \text{if } p = 1^k.i.q \text{ and } 1 \le i \le n. \end{cases}$$

  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 $\mathit{MPos}(t)$.
- *The sequence of variables abstracted on the path to position $p \in \mathit{Pos}(t)$:*

$$\mathit{seq}_{bv}(t,p) = \begin{cases} \epsilon & \text{if } p = \epsilon, \\ x.\mathit{seq}_{bv}(s,q) & \text{if } t = \lambda x.s \text{ and } p = 1.q, \\ \mathit{seq}_{bv}(t_i,q) & \text{if } t = a(\overline{t_n}),\ 0 < i \le n, \text{ and } p = i.q. \end{cases}$$

  The set of *variables abstracted* on the path to position $p \in \mathit{Pos}(t)$ is $\mathcal{BV}(t,p) = \{\mathit{seq}_{bv}(t,p)\}$, and the set of *variables with bound occurrences* in t is $\mathcal{BV}(t) = \bigcup_{p \in \mathit{Pos}(t)} \mathcal{BV}(t,p)$. Moreover, we also define $t\restriction_p = \lambda\overline{x_k}.(t|_p)$, where $\overline{x_k} = \mathit{seq}_{bv}(t,p)$.

A *pattern* [9] 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 [9]. 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 $f(\overline{l_n}) \to r \Leftarrow C$, where*

- *$f(\overline{l_n})$ 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 $l_1, \ldots, l_n \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$.

Finally, we also find it convenient to define the *$\overline{x_k}$-lifter* of a term t:

Definition 2 (Lifter). *Given a term t, a subset V of $\mathcal{FV}(t)$, and a sequence $\overline{x_k}$ of distinct variables with no occurrences in t, the* $\overline{x_k}$-lifter of t with respect to V *is the term $t^{\uparrow \overline{x_k} \restriction V}$, defined recursively as follows:*

$$t^{\uparrow \overline{x_k} \restriction V} = \begin{cases} \lambda \overline{y_l}.(\pi^{\uparrow (\overline{y_l}, \overline{x_k}) \restriction V}) & \text{if } t \equiv \lambda \overline{y_l}.\pi, \\ a\left(\overline{t_n^{\uparrow \overline{x_k} \restriction V}}\right) & \text{if } t \equiv a(\overline{t_n}) \text{ with } a \notin V, \\ X\left(\overline{x_k}, \overline{t_n^{\uparrow \overline{x_k} \restriction V}}\right) & \text{if } t \equiv X(\overline{t_n}) \text{ with } X \in V. \end{cases}$$

*The $\overline{x_k}$-*lifter of a term t *is the term $t^{\uparrow \overline{x_k}} = t^{\uparrow \overline{x_k} \restriction \mathcal{FV}(t)}$. We also define $t^{\updownarrow \overline{x_k}} = \lambda \overline{x_k}.(t^{\uparrow \overline{x_k}})$. If $C \equiv \overline{s_m == t_m}$ is a sequence of equations then we write $C^{\updownarrow \overline{x_k}}$ for the sequence $\overline{s_m^{\updownarrow \overline{x_k}} == t_m^{\updownarrow \overline{x_k}}}$.*

3 The Higher-Order Rewriting Logic GHRC

In this section we extend the constructor-based Conditional ReWriting Logic CRWL from [3, 4], in order to deal with conditional pattern rewrite rules. In contrast to Meseguer's rewriting logic [8], 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 [3, 4], we do not impose non-ambiguity conditions. This means that non-deterministic functions are allowed.

For all these reasons, we want 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, 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. First, we need to define the suitable notion of *value* that is used in our setting with λ-abstractions and higher-order unification.

Definition 3 (Values). *A* value *is a partial term t which has the following property:*

$$\forall\, p \in MPos(t),\ \forall\, (\pi \to r \Leftarrow C) \in \mathcal{R} :\ \nexists\, matcher(t|_p, \pi^{\updownarrow seq_{bv}(t,p)})$$

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 $Val(\mathcal{F}_\bot, \mathcal{V})$ (resp., $Val(\mathcal{F}, \mathcal{V})$) for the set of values (resp., total values), and $VSubst(\mathcal{F}_\bot, \mathcal{V})$ 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 $s, t \in \mathcal{T}(\mathcal{F}_\bot, \mathcal{V})$ 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 4.
- *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 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 [3, 4]). 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 [3], Section 3). The other inference rules in GHRC are easier to understand.

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*	$\dfrac{\lambda\overline{x_k}.s_1 \twoheadrightarrow l_1^{\downarrow\overline{x_k}}\theta \cdots \lambda\overline{x_k}.s_n \twoheadrightarrow l_n^{\downarrow\overline{x_k}}\theta \qquad \dfrac{C^{\downarrow\overline{x_k}}\theta \qquad r^{\downarrow\overline{x_k}}\theta \twoheadrightarrow u}{\boxed{\lambda\overline{x_k}.f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \twoheadrightarrow u}}}{\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u}$ if $u \neq \lambda\overline{x_k}.\bot$, $\theta \in \mathit{VSubst}(\mathcal{F}_\bot, \mathcal{V})$, 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 $u \in \mathit{Val}(\mathcal{F}, \mathcal{V})$.

Table 1. The GHRC proof calculus.

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:

$$\textbf{AR} \quad \frac{\lambda\overline{x_k}.s_1 \twoheadrightarrow l_1^{\downarrow\overline{x_k}}\theta \quad \cdots \quad \lambda\overline{x_k}.s_n \twoheadrightarrow l_n^{\downarrow\overline{x_k}}\theta \qquad \lambda\overline{x_k}.f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \twoheadrightarrow u}{\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u}$$

if $f \in \mathcal{F}_d$, $u \neq \lambda\overline{x_k}.\bot$, and $\theta \in \mathit{VSubst}(\mathcal{F}_\bot, \mathcal{V})$.

$$\textbf{FA} \quad \frac{C^{\downarrow\overline{x_k}}\theta \qquad r^{\downarrow\overline{x_k}}\theta \twoheadrightarrow u}{\lambda\overline{x_k}.f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \twoheadrightarrow u}$$

if $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$ and $\theta \in \mathit{VSubst}(\mathcal{F}_\bot, \mathcal{V})$.

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 5.

Detailed examples of GHRC-derivations in the form of *proof trees* in this kind of rewriting logics can be found in [3, 4, 11] 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 4 (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:* $s == t \equiv t == s$.
- $\gamma \in \mathit{Subst}(\mathcal{F}_\perp, \mathcal{V})$ *is a* solution *of a goal* $G \equiv \{\!\{\overline{s_n == t_n}\}\!\}$ *if* $\gamma\restriction_{\mathcal{FV}(G)} \in \mathit{VSubst}(\mathcal{F}_\perp, \mathcal{V})$*, 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 $\mathit{Soln}(G)$ *for the set of solutions of a goal* G.

Example 1. For the particular function $f \rightarrow \lambda xs.\ \mathit{pair}(\mathit{sum}(xs),\ \mathit{length}(xs))$, where *pair* is a data constructor, and

$$\begin{array}{lll} \mathit{sum}\,([\,]) \rightarrow 0 & \mathit{length}\,([\,]) \rightarrow 0 & \mathit{fst}\,(\mathit{pair}\,(x,y)) \rightarrow x \\ \mathit{sum}\,([x|xs]) \rightarrow x{+}\mathit{sum}(xs) & \mathit{length}\,([x|xs]) \rightarrow 1{+}\mathit{length}(xs) & \mathit{snd}\,(\mathit{pair}(x,y)) \rightarrow y \end{array}$$

we can check that $\gamma = \{E \mapsto \mathit{pair}(0,0), G \mapsto \lambda u, z.\mathit{pair}(u + \mathit{fst}(z), 1 + \mathit{snd}(z))\}$ is a solution of the goal $\{\!\{f([\,]) == E,\ \lambda x, xs.\, f([x|xs]) == \lambda x, xs.\, G(x,\ f(xs))\}\!\}$. For example, we have the following logical proof in the GHRC-calculus for $\mathcal{R} \vdash \lambda x, xs.\ f([x|xs]) == \lambda x, xs.\ \mathit{pair}(x + \mathit{fst}(f(xs)), 1 + \mathit{snd}(f(xs)))$, where $\mathcal{R}$ is a CPRS containing all the pattern rewrite rules mentioned in this example.

$$\begin{array}{l}
\mathbf{J}\ \lambda x, xs.\ f([x|xs]) == \lambda x, xs.\ \mathit{pair}(x + \mathit{fst}(f(xs)), 1 + \mathit{snd}(f(xs))) \\
\quad \mathbf{OR}\ \lambda x, xs.\ f([x|xs]) \twoheadrightarrow \lambda x, xs.\ \mathit{pair}(x + \mathit{sum}(xs), 1 + \mathit{length}(xs)) \\
\qquad \mathbf{RF}\ \lambda x, xs.\ [x|xs] \twoheadrightarrow \lambda x, xs.\ [x|xs] \\
\qquad \mathbf{MN}\ \lambda x, xs.\ \mathit{pair}(\mathit{sum}([x|xs]), \mathit{length}([x|xs])) \twoheadrightarrow \\
\qquad\qquad\qquad\qquad \lambda x, xs.\ \mathit{pair}(x + \mathit{sum}(xs), 1 + \mathit{length}(xs)) \\
\qquad\quad \mathbf{OR}\ \lambda x, xs.\ \mathit{sum}([x|xs]) \twoheadrightarrow \lambda x, xs.\ (x + \mathit{sum}(xs)) \\
\qquad\qquad \mathbf{RF}\ \lambda x, xs.\ [x|xs] \twoheadrightarrow \lambda x, xs.\ [x|xs] \\
\qquad\qquad \mathbf{RF}\ \lambda x, xs.\ (x + \mathit{sum}(xs)) \twoheadrightarrow \lambda x, xs.\ (x + \mathit{sum}(xs)) \\
\qquad\quad \mathbf{OR}\ \lambda x, xs.\ \mathit{length}([x|xs]) \twoheadrightarrow \lambda xs.\ (1 + \mathit{length}(xs)) \\
\qquad\qquad \mathbf{RF}\ \lambda x, xs.\ [x|xs] \twoheadrightarrow \lambda x, xs.\ [x|xs] \\
\qquad\qquad \mathbf{RF}\ \lambda xs.\ (1 + \mathit{length}(xs)) \twoheadrightarrow \lambda xs.\ (1 + \mathit{length}(xs))
\end{array}$$

$$
\begin{array}{l}
\textbf{MN}\ \lambda s, xs.\ pair(x + fst(f(xs)), 1 + snd(f(xs))) \twoheadrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda x, xs.\ pair(x + sum(xs), 1 + length(xs)) \\
\quad \textbf{MN}\ \lambda xs.\ (x + fst(f(xs))) \twoheadrightarrow \lambda x, xs.\ (x + sum(xs)) \\
\quad\quad \textbf{RF}\ \lambda x.\ x \twoheadrightarrow \lambda x.\ x \\
\quad\quad \textbf{OR}\ \lambda xs.\ fst(f(xs)) \twoheadrightarrow \lambda xs.\ sum(xs) \\
\quad\quad\quad \textbf{OR}\ \lambda xs.\ f(xs) \twoheadrightarrow \lambda xs.\ pair(sum(xs), length(xs)) \\
\quad\quad\quad\quad \textbf{RF}\ \lambda xs.\ xs \twoheadrightarrow \lambda xs.\ xs \\
\quad\quad\quad\quad \textbf{RF}\ \lambda xs.\ pair(sum(xs), length(xs)) \twoheadrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda xs.\ pair(sum(xs), length(xs)) \\
\quad\quad\quad \textbf{RF}\ \lambda xs.\ sum(xs) \twoheadrightarrow \lambda xs.\ sum(xs) \\
\quad \textbf{MN}\ \lambda xs.\ (1 + snd(f(xs))) \twoheadrightarrow \lambda xs.\ (1 + length(xs)) \\
\quad\quad \textbf{RF}\ 1 \twoheadrightarrow 1 \\
\quad\quad \textbf{OR}\ \lambda xs.\ snd(f(xs)) \twoheadrightarrow \lambda xs.\ length(xs) \\
\quad\quad\quad \textbf{OR}\ \lambda xs.\ f(xs) \twoheadrightarrow \lambda xs.\ pair(sum(xs), length(xs)) \\
\quad\quad\quad\quad \textbf{RF}\ \lambda xs.\ xs \twoheadrightarrow \lambda xs.\ xs \\
\quad\quad\quad\quad \textbf{RF}\ \lambda xs.\ pair(sum(xs), length(xs)) \twoheadrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda xs.\ pair(sum(xs), length(xs)) \\
\quad\quad\quad \textbf{RF}\ \lambda xs.\ length(xs) \twoheadrightarrow \lambda xs.\ length(xs)
\end{array}
$$

□

Finally, we give a result which characterizes the semantics proofs built with GHRC and generalizes useful known properties of CRWL-deductions for the first-order case (see [3, 4, 11] for more details). The proof is given in [14].

Lemma 1 (Basic Semantic Property of GHRC-deductions). *Let $s \in Val(\mathcal{F}_\bot, \mathcal{V})$. If $\mathcal{R} \vdash s \twoheadrightarrow t$ then $t \in Val(\mathcal{F}_\bot, \mathcal{V})$, $s \sqsupseteq t$, and $\mathcal{R} \nvdash_{\mathbf{OR}} s \twoheadrightarrow t$. Moreover, if $t \in Val(\mathcal{F}, \mathcal{V})$ then $s \equiv t$.*

4 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 paper. The semantic definition of *interpretation* is simpler than the one in the first-order setting [3, 4, 14], where a more general notion of interpretation (under the name of *Algebra*) is presented. In our debugging scheme we will assume that the *intended model* of a CPRS is an interpretation.

Definition 5 (Interpretations and Models).

(1) A basic fact $\lambda\overline{x_k}.\, f(\overline{t_n^{\downarrow \overline{x_k}}}) \twoheadrightarrow u$ 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* $\overline{t_n}, \overline{s_n} \in Val(\mathcal{F}_\perp, \mathcal{V})$ *arbitrary partial terms of the same base type that* $t, s \in Val(\mathcal{F}_\perp, \mathcal{V})$*:*

- $(\lambda\overline{x_k}.\, f(\overline{t_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow \lambda\overline{x_k}.\, \perp) \in \mathcal{I}$.
- *If* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow \lambda\overline{x_k}.\, t) \in \mathcal{I}$*,* $\lambda\overline{x_k}.\, t_i^{\downarrow\overline{x_k}} \sqsubseteq \lambda\overline{x_k}.\, s_i^{\downarrow\overline{x_k}}$*,* $\lambda\overline{x_k}.\, t \sqsupseteq \lambda\overline{x_k}.\, s$*, then also* $(\lambda\overline{x_k}.\, f(\overline{s_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow \lambda\overline{x_k}.\, s) \in \mathcal{I}$.
- *If* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow \lambda\overline{x_k}.\, t) \in \mathcal{I}$ *and* $\theta \in VSubst(\mathcal{F}_\perp, \mathcal{V})$*, then* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\downarrow\overline{x_k}}}\theta) \twoheadrightarrow \lambda\overline{x_k}.\, t\theta) \in \mathcal{I}$.

(3) *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* $GHRC_{\mathcal{I}}$*, consisting of the GHRC rules* **B**, **MN**, **RF** *and* **J** *together with the inference rule* $\mathbf{OR}_{\mathcal{I}}$*:*

$$\mathbf{OR}_{\mathcal{I}} \quad \frac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow t_1^{\downarrow\overline{x_k}} \;\cdots\; \lambda\overline{x_k}.\, s_n \twoheadrightarrow t_n^{\downarrow\overline{x_k}} \qquad u \twoheadrightarrow s}{\lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow s}$$

if $u \neq \lambda\overline{x_k}.\, \perp$ *and* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow u) \in \mathcal{I}$.

In general, for every basic fact $\lambda\overline{x_k}.\, f(\overline{t_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow u$*, it can be proved that it is valid in* $\mathcal{I}$ *iff* $(\lambda\overline{x_k}.\, f(\overline{t_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow u) \in \mathcal{I}$.

(4) *The denotation of a term* $t \in \mathcal{T}(\mathcal{F}_\perp, \mathcal{V})$ *is the set:*

$$[\![\, t \,]\!]^{\mathcal{I}} = \{\, s \in Val(\mathcal{F}_\perp, \mathcal{V}) \mid t \twoheadrightarrow s \text{ is valid in } \mathcal{I} \,\}$$

(5) $\mathcal{I}$ *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* $\theta \in VSubst(\mathcal{F}_\perp, \mathcal{V})$ *and* $C \equiv \overline{s_m == t_m}$*, either*

- $[\![\, s_i\theta \,]\!]^{\mathcal{I}} \cap [\![\, t_i\theta \,]\!]^{\mathcal{I}} \cap Val(\mathcal{F}, \mathcal{V}) \neq \varnothing$ *(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 5 we can prove that the GHRC proof calculus is semantically sound.

Theorem 1 (Semantic Correctness of GHRC). *If* $G \equiv \{\!\{\overline{s_n == t_n}\}\!\}$ *is a goal for a CPRS* $\mathcal{R}$ *and* $\gamma \in Soln(G)$ *then* $\gamma \in Soln_{\mathcal{I}}(G)$ *for all model* $\mathcal{I}$ *of* $\mathcal{R}$ *(i.e., every* $s_i\gamma == t_i\gamma$ *is valid in* $\mathcal{I}$*).*

Proof. The proof is quite standard and details can be found in [14]: It is sufficient to assume an arbitrarily given model $\mathcal{I} \models \mathcal{R}$ and to prove that any GHRC inference rule whose premises are valid in $\mathcal{I}$ has a conclusion that is also valid in $\mathcal{I}$. □

5 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 6 (Symptoms and Errors). *Assume that $\mathcal{I}$ is the intended model for a given CPRS $\mathcal{R}$, and consider a substitution $\gamma \in VSubst(\mathcal{F}, \mathcal{V})$ 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 $\gamma \notin Soln_{\mathcal{I}}(G)$ (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 $\gamma \in Soln(G)$. The goal solving calculus HOLN^{DT} given in [13] is GHRC-sound. This claim can be proved by a straightforward adaptation of the *soundness theorem* for HOLN^{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 $\gamma \in Subst(\mathcal{F}, \mathcal{V})$ 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 $\mathcal{I}$.*

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 6. Therefore, we can conclude that the user's intended model $\mathcal{I}$ is not a model of $\mathcal{R}$. Then, by Definition 5, some conditional pattern rewrite rule belonging to $\mathcal{R}$ is not valid in $\mathcal{I}$. □

The debugging scheme proposed in [10] 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 [10], an erroneous node with no erroneous children in 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:

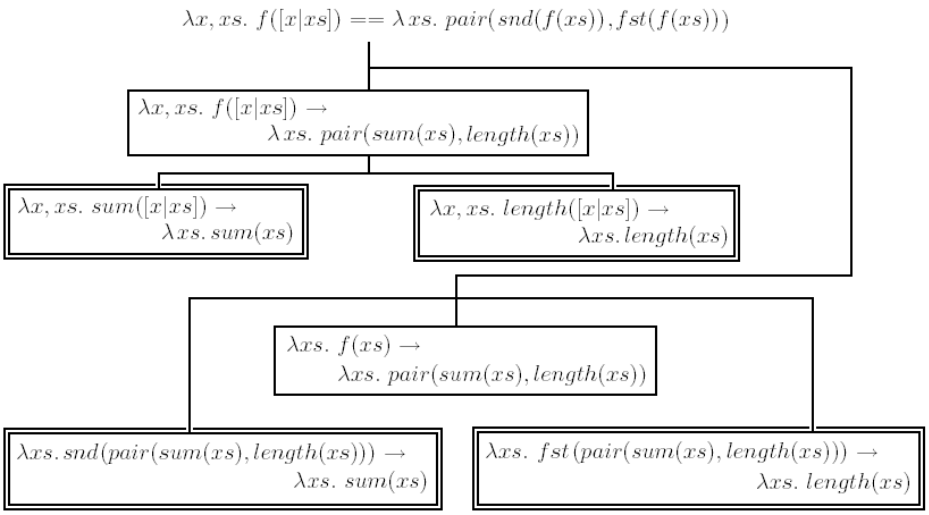


Fig. 2. Computation tree in GHRC for declarative debugging

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 [10], 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 $\lambda\overline{x_k}.\, f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \twoheadrightarrow u$ 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 $(f(\overline{l_n}) \rightarrow r \Leftarrow C) \in \mathcal{R}$ and $\theta \in VSubst(\mathcal{F}_\perp, \mathcal{V})$. Therefore, the children of $\lambda\overline{x_k}.\, f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \twoheadrightarrow u$ in the GHRC-proof tree, $C^{\downarrow\overline{x_k}}\theta$ and $r^{\downarrow\overline{x_k}}\theta \twoheadrightarrow u$ are valid in $\mathcal{I}$, because they are the children of a buggy node. With this we can conclude that $C^{\downarrow\overline{x_k}}\theta$ and $r^{\downarrow\overline{x_k}}\theta \twoheadrightarrow u$ are valid in $\mathcal{I}$ (i.e., $\mathcal{I}$ satisfies $C^{\downarrow\overline{x_k}}\theta$ and $u \in [\![\, r^{\downarrow\overline{x_k}}\theta \,]\!]^{\mathcal{I}}$), while $\lambda\overline{x_k}.\, f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \twoheadrightarrow u$ is not valid in $\mathcal{I}$ (i.e., $u \notin [\![\, \lambda\overline{x_k}.\, f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \,]\!]^{\mathcal{I}}$). Then $[\![\, r^{\downarrow\overline{x_k}}\theta \,]\!]^{\mathcal{I}} \not\subseteq [\![\, \lambda\overline{x_k}.\, f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \,]\!]^{\mathcal{I}}$, which means (see Definition 5) that the conditional pattern rewrite rule $(f(\overline{l_n}) \rightarrow r \Leftarrow C) \in \mathcal{R}$ is not valid in $\mathcal{I}$. □

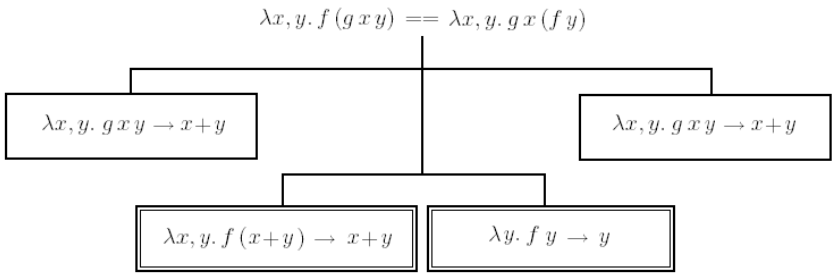


Fig. 3. Another computation tree in GHRC for declarative debugging

Example 2. Consider again the setting of *Example 1*, and the same goal $\{\!\{\lambda x, xs.\ f([x|xs]) == \lambda x, xs.\, G(x,\ f(xs))\}\!\}$. Now 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 shown in *Example 1*, and the right computed answer for the previous goal if we repeat the computation. □

Example 3. Consider the simple property $(+1) \circ foldr\ (+)\ 0 = foldr\ (+)\ 1$, involving the classical fold-right function *foldr*, the composition of function $\circ$, and functions $(+)$, $(+1)$ to sum natural numbers. We want to prove this result by applying the following well-known theorem on lambda abstractions called *fusion law* [6]: $\lambda x, y.\, f\,(g\,x\,y) = \lambda x, y.\, h\,x\,(f\,y) \Rightarrow f \circ foldr\ g\,z = foldr\ h\,(f\,z)$. For this purpose, we try to identify correctly functions f, g, h, z in order to compound an appropriate CPRS $\mathcal{R}$ and to apply this result, but we make a mistake: $f \to \lambda z.\, z$ (instead of $f \to \lambda z.\, z + 1$), $g \to \lambda u, v.\ u + v$, and $z \to 0$. Now, we can compute h by applying a GHRC-sound goal solving system to the goal $G = \{\!\{\lambda x, y.\, f\,(g\,x\,y) == \lambda x, y.\, H\,x\,(f\,y)\}\!\}$. We obtain the computed answer $\gamma = \{H \mapsto \lambda u, v.\ u + v\}$ (or equivalently, as a rewrite rule, $h \to g$). Then, we have completed the CPRS $\mathcal{R}$ and we can apply the theorem, but we obtain only a trivial identity $foldr(+)\,0 = foldr(+)\,0$ instead of our initial property. To find the bug we examine the computation tree from the GHRC-proof tree for $\gamma \in Soln(G)$ (see **Fig. 3**). There is at least a *buggy node* labeled with the basic fact $\lambda y.\, f\,y \to y$ which is not valid in the user's intended model, because the user knows that $f\,0 \to 1$ instead of $f\,0 \to 0$. Therefore, f is not well defined in the CPRS $\mathcal{R}$ and the user can rewrite it, putting $f \to \lambda z.\, z + 1$. □

6 Conclusions and Future Work

We have presented a generalization of the declarative method for diagnosing wrong computed answers in first-order lazy functional logic programs to the more expressive setting of the simply typed λ-calculus, where the notion of lazy and possibly non-deterministic higher-order function plays a central role.

Planned future work will include further theoretical investigation to integrate non-equality constraints (e.g., see $-\pi/2 \leq f(x) \leq \pi/2$ and $f(x) \neq 0$ in Section 1) in the conditional part of pattern rewrite rules, following the line of recent researches on *constraint rewriting logics* for declarative programming [12].

References

1. R. Caballero, F.J. López-Fraguas, and M. Rodríguez-Artalejo. *Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs*. In Proc. FLOPS'01, LNCS vol. 2024, pp. 170–184, 2001.
2. J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. *A higher-order rewriting logic for functional logic programming*. In Proc. ICLP'97, pp. 153–167, 1997.
3. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. *A Rewriting Logic for Declarative Programming*. In Proc. ESOP'96, LNCS vol. 1058, pp. 156–172, 1996.
4. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. *An approach to declarative programming based on a rewriting logic*. *Journal of Logic Programming*, 40:47–87, 1999.
5. M. Hanus and C. Prehofer. *Higher-order narrowing with definitional trees*. Journal of Functional Programming, 9(1): 33–75, 1999.
6. J.R. Hindley and J.P. Seldin. *Introduction to Combinatorics and λ-Calculus*. Cambridge University Press, 1986.
7. F.J. López-Fraguas and J. Sánchez-Hernández. *$\mathcal{TOY}$: A Multiparadigm Declarative System*. In Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999. System and documentation available at `http://toy.sourceforge.net`.
8. J. Meseguer. *Conditional rewriting logic as a unified model of concurrency*. TCS 96, pp. 73–155, 1992.
9. D. Miller. *A logic programming language with λ-abstraction, function variables, and simple unification*. Journal of Logic and Computation, 1(4):497–536, 1991.
10. L. Naish. *A Declarative Debugging Scheme*. Journal of Functional and Logic Programming, 1997-3.
11. R. del Vado-Vírseda. *A Demand-driven Narrowing Calculus with Overlapping Definitional Trees*. In Proc. PPDP'03, ACM pp. 253–263, 2003.
12. R. del Vado-Vírseda. *Declarative Constraint Programming with Definitional Trees*. In Proc. FroCoS'05, LNAI vol. 3717, pp. 184–199, 2005.
13. R. del Vado-Vírseda. *A Higher-Order Demand-Driven Narrowing Calculus with Definitional Trees*. In Proc. ICTAC'07, LNCS vol. 4711, pp. 169–184, 2007.
14. R. del Vado-Vírseda. *A Higher-Order Rewriting Logic on Lambda Abstractions for Declarative Programming*. Technical Report, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, April 2009. available at `http://www.fdi.ucm.es/profesor/rdelvado/TR09.pdf`.

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

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 [10] 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, 7], and delimited continuations. Hasegawa [8] 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 [17], in which he showed that the negation and conjunction fragment of a lambda calculus suffices for a CPS calculus as the target of various first-order calculi.

Domain-free type systems [1], which are in an intermediate style between Church and Curry style, are useful for studying 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

* yuki@kuis.kyoto-u.ac.jp
** knak@kuis.kyoto-u.ac.jp

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 [6], 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 [13], 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 that asks whether $\Gamma \vdash M : A$ is derivable for given Γ, M, and A. *Type-inference problem* (TI) is another problem that asks 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 [18] 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 [12] 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 [16]. TC and TI in domain-free lambda calculi with existential types were proved to be undecidable in [11, 12]. 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 [12]. 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, we need a non-trivial idea to prove the reduction from TC to TC_0 since DF-$\lambda^{\neg\wedge\exists}$ does not have implication. 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 DF-$\lambda^{\neg\wedge\exists}$. 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. [9] and Schubert [14], 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 DF-$\lambda^{\to\exists}$ and DF-$\lambda^{\neg\wedge\exists}$.

Curry style:	SUP	$\overset{[18]}{\leq}$	TC in F	$\overset{[18]}{\simeq}$	TI in F
domain-free style:	2UP	$\overset{[4]}{\leq}$	TC in DF-F	$\overset{[12]}{\simeq}$	TI in DF-F
			$\mid\wedge$ [11]		$\mid\wedge$ [11]
domain-free style:			TC in DF-$\lambda^{\to\exists}/\lambda^{\neg\wedge\exists}$	$\simeq^*$	TI in DF-$\lambda^{\to\exists}/\lambda^{\neg\wedge\exists}$

Fig. 1. TC and TI for polymorphic and existential types

The section 2 introduces the domain-free lambda calculi with existence: DF-$\lambda^{\to\exists}$ and DF-$\lambda^{\neg\wedge\exists}$. 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: DF-$\lambda^{\to\exists}$ and DF-$\lambda^{\neg\wedge\exists}$.

As pointed out in [10], 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 $\texttt{let}\ \langle X, x\rangle = N\ \texttt{in}\ P$. In this paper, we use the notation $M[Xx.N]$ following [11, 12]. We can write the term $\langle A, M\rangle$ in the style of modules of Standard ML as $\texttt{struct type}\ X = A\ \texttt{val}\ x = M\ \texttt{end}$.

2.1 Lambda Calculus with Implication and Existence

First, we define the domain-free lambda calculus $\mathsf{DF}\text{-}\lambda^{\to\exists}$ 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 $\mathsf{DF}\text{-}\lambda^{\to\exists}$ 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 $M_1M_2\cdots M_n$ denotes $((M_1M_2)\cdots)M_n$. 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 that asks whether $\Gamma \vdash M : A$ is derivable for given Γ, M, and A. *Type inference* (TI) is a problem that asks 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 [18], and in the domain-free polymorphic lambda calculus [12]. 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 [18, 12], 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 [11, 12] 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 DF-$\lambda^{\rightarrow\exists}$ and DF-$\lambda^{\neg\wedge\exists}$.

Theorem 1. *1. Type checking and type inference are equivalent in* DF-$\lambda^{\rightarrow\exists}$*, that is, type checking in* DF-$\lambda^{\rightarrow\exists}$ *is Turing reducible to type inference in* DF-$\lambda^{\rightarrow\exists}$ *and vice versa.*

2. Type checking and type inference are equivalent in DF-$\lambda^{\neg\wedge\exists}$*.*

This result gives another proof of undecidability of TI in these calculi since TC is undecidable 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 [12] for polymorphic types.

In DF-$\lambda^{\rightarrow\exists}$, the reduction from TC to TC_0 is easy, whereas the reduction is not easy to prove for DF-$\lambda^{\neg\wedge\exists}$ due to absence of implication. In our proof, we show that we can construct a DF-$\lambda^{\neg\wedge\exists}$-term $\underline{\lambda}x.M$ for a DF-$\lambda^{\neg\wedge\exists}$-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 DF-$\lambda^{\neg\wedge\exists}$.

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 DF-$\lambda^{\neg\wedge\exists}$, 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 DF-$\lambda^{\rightarrow\exists}$ 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 DF-$\lambda^{\neg\wedge\exists}$.

4 TC and TI are Equivalent in DF-$\lambda^{\rightarrow\exists}$

In this section, we prove that TC and TI are equivalent in DF-$\lambda^{\rightarrow\exists}$.

At first, we show that TI is Turing reducible to TC (that is denoted by TI $\leq$ TC).

Proposition 1. *TI in* DF-$\lambda^{\rightarrow\exists}$ *is Turing reducible to TC in* DF-$\lambda^{\rightarrow\exists}$*.*

Proof. For a given instance $? \vdash M :?$ of TI in DF-$\lambda^{\rightarrow\exists}$, 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 $X \to X$, then M has some type. Conversely, if $\Gamma \vdash M : A$ holds for some Γ and A, we have the following for some B,

$$\dfrac{\dfrac{\dfrac{}{y : X, x : B \vdash y : X}\ (Ax)}{y : X \vdash \lambda x.y : B{\to}X}\ (\to I) \qquad \begin{array}{c}\Gamma \vdash M : A \\ \vdots \\ \vdash \lambda x_1 \cdots \lambda x_n.M : B\end{array}}{\dfrac{y : X \vdash (\lambda x.y)(\lambda x_1 \cdots \lambda x_n.M) : X}{\vdash \lambda y.(\lambda x.y)(\lambda x_1 \cdots \lambda x_n.M) : X{\to}X}\ (\to I),}\ (\to E)$$

where we can suppose that $\{x_1, \cdots, x_n\} = \{z \mid (z : C) \in \Gamma\}$ without loss of generality since 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 $\leq$ TI consists of two steps: TC $\leq$ TC_0 and $\mathrm{TC}_0 \leq$ TI. It is easy to prove TC $\leq$ TC_0 for DF-$\lambda^{\to\exists}$. So, in the following, we show $\mathrm{TC}_0 \leq$ TI. More concretely, we show that, for a given instance $\vdash M : A?$ of TC_0, we can effectively construct a DF-$\lambda^{\to\exists}$-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 $A{\to}O$.

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 $C{\to}D$, $\mathsf{ldep}((C{\to}D)[Y := B]) = \mathsf{ldep}((C[Y := B]){\to}(D[Y := B])) = \mathsf{ldep}(C[Y := B]) + 1$ holds. Furthermore, $\mathsf{ldep}(C{\to}D) = \mathsf{ldep}(C) + 1$ holds. So 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 $\mathsf{lvar}(C) \equiv Y$. So $\mathsf{lvar}(C{\to}D) \equiv Y$ holds.

When A is $\exists X.C$, $\mathsf{ldep}((\exists X.C)[Y := B]) = \mathsf{ldep}(C[Y := B]) + 1$ holds. Furthermore, $\mathsf{ldep}(\exists X.C) = \mathsf{ldep}(C) + 1$ holds. So 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 $\mathsf{lvar}(C) \equiv Y$. So $\mathsf{lvar}(\exists X.C) \equiv Y$ holds.

2. By induction on A.

When A is a variable, A is X since $\mathsf{lvar}(A) \equiv X$ holds. 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 $\mathsf{lbdep}((C{\to}D)[Y := B]) = \mathsf{lbdep}(C[Y := B]{\to}D[Y := B]) = \mathsf{lbdep}(C[Y := B]) = \mathsf{lbdep}(B)$ holds. Moreover, from the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B)$ and so we have $\mathsf{lvar}((C{\to}D)[Y := B]) \equiv \mathsf{lvar}(C[Y := B]{\to}D[Y := B]) \equiv \mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B)$.

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 bound variable of A. From the induction hypothesis, we have $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B) \not\equiv X$, and then we have $\mathsf{lbdep}((\exists X.C)[Y := B]) = \mathsf{lbdep}(\exists X.C[Y := B]) = \mathsf{lbdep}(C[Y := B])$. From the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lbdep}(C[Y := B]) = \mathsf{lbdep}(B)$. Hence we have $\mathsf{lbdep}((\exists X.C)[Y := B]) = \mathsf{lbdep}(B)$. Moreover from the induction hypothesis, $\mathsf{lvar}(C) \equiv Y$ implies $\mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B)$ and so we have $\mathsf{lvar}((\exists X.C)[Y := B]) \equiv \mathsf{lvar}(\exists X.C[Y := B]) \equiv \mathsf{lvar}(C[Y := B]) \equiv \mathsf{lvar}(B)$ 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.

$$\dfrac{\dfrac{}{\Gamma \vdash x : \exists Y.C{\to}A}\ (Ax) \quad \dfrac{\dfrac{}{\Gamma \vdash x : C[X := \neg_O\exists X.X]}\ (Ax)}{\Gamma \vdash \langle\neg_O\exists X.X, x\rangle : \exists Y.C}\ (\exists I)}{\Gamma \vdash x\langle\neg_O\exists X.X, x\rangle : A}\ (\to E)$$

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, we have

$$\mathsf{lbdep}(\exists Y.C{\to}A) = \mathsf{lbdep}(\exists Y.C) = \mathsf{ldep}(C)$$

since $\mathsf{lvar}(C) \equiv Y$ holds. Therefore, we have $\mathsf{ldep}(C) = 0$, and then C must be a variable. Hence C is identical to Y since $\mathsf{lvar}(C) \equiv Y$ holds, and so C_x must be $\neg_O\exists X.X$. □

Then we can show the following proposition, from which $\mathrm{TC}_0 \leq \mathrm{TI}$ follows directly.

Proposition 2. *For a closed* DF-$\lambda^{\to\exists}$*-term* M *and a* $\to\exists$*-type* A*, we can effectively construct a closed* DF-$\lambda^{\to\exists}$*-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 $\vdash J_{M,A} : \neg_O\neg_O\exists X.X$, since $x : \neg_O\exists X.X \vdash x\langle A, M\rangle : O$ is derivable as follows.

$$\dfrac{\dfrac{}{x : \neg_O\exists X.X \vdash x : \neg_O\exists X.X}\ (Ax) \quad \dfrac{\begin{array}{c}\vdots\\ \vdash M : A\end{array}}{\vdash \langle A, M\rangle : \exists X.X}\ (\exists I)}{x : \neg_O\exists X.X \vdash x\langle A, M\rangle : O}\ (\to E)$$

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. $\mathrm{TC}_0 \leq$ 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-$\lambda^{\to\exists}$, and consists of the following four parts: (i) TC $\leq$ TC_0, (ii) $\mathrm{TC}_0 \leq$ TI, (iii) TI $\leq$ TI_0, and (iv) $\mathrm{TI}_0 \leq$ TC. In contrast to DF-$\lambda^{\to\exists}$, 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$$\mathrm{TC}_0$ and TI$\leq$$\mathrm{TI}_0$. In DF-$\lambda^{\neg\wedge\exists}$, we cannot type the term $\lambda x_1.\cdots\lambda x_n.M$, since N has to be typed with $\bot$ in order to type the lambda abstraction $\lambda x.N$. So 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 $\to$, $A_1{\Rightarrow}\cdots{\Rightarrow}A_n{\Rightarrow}B$ 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 $\Gamma\vdash\underline{\lambda}x.M:\neg(A\wedge\neg B)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{c:A\wedge\neg B\vdash c:A\wedge\neg B}}{c:A\wedge\neg B\vdash c\pi_2:\neg B}\quad\begin{array}{c}\vdots\\ \Gamma, x:A\vdash M:B\end{array}}{\Gamma, c:A\wedge\neg B, x:A\vdash(c\pi_2)M:\bot}}{\Gamma, c:A\wedge\neg B\vdash\lambda x.(c\pi_2)M:\neg A}\quad\dfrac{\dfrac{}{c:A\wedge\neg B\vdash c:A\wedge\neg B}}{c:A\wedge\neg B\vdash c\pi_1:A}}{\Gamma, c:A\wedge\neg B\vdash(\lambda x.(c\pi_2)M)(c\pi_1):\bot}}{\Gamma\vdash\underline{\lambda}x.M:\neg(A\wedge\neg B)}$$

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 DF-$\lambda^{\neg\wedge\exists}$ is Turing reducible to TC_0 in DF-$\lambda^{\neg\wedge\exists}$.*

2. TI in DF-$\lambda^{\neg\wedge\exists}$ is Turing reducible to TI_0 in DF-$\lambda^{\neg\wedge\exists}$.

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 since 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 $A \to B$. 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 $A \to B$ is (classically) equivalent to $\neg(A \wedge \neg B)$, which is denoted by $A \Rightarrow B$ in this paper. $A \Rightarrow B$ is not an intuitionistic implication, since we cannot conclude B from $A \Rightarrow B$ and A in the intuitionistic logic. 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, since 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 [17, 5, 7].

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:

$$N \equiv \lambda c.(\lambda x_1.\cdots(\lambda x_{n-1}.(\lambda x_n.(c\pi_{n+1}^{n+1})M)(c\pi_n^{n+1}))(c\pi_{n-1}^{n+1})\cdots)(c\pi_1^{n+1}) : \bot,$$
$$C \equiv \neg(A_1 \wedge \cdots \wedge A_n \wedge \neg B),$$

where π_m^n 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 $\mathsf{DF}\text{-}\lambda^{\to\exists}$ 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 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. □

The following proposition is also proved similarly to $\mathsf{DF}\text{-}\lambda^{\to\exists}$.

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. □

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 $\mathsf{DF}\text{-}\lambda^{\to\exists}$, 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. □

6 Concluding Remarks

In this paper, we show equivalence between type checking and type inference in the domain-free lambda calculi with existential types: $\mathsf{DF}\text{-}\lambda^{\to\exists}$ and $\mathsf{DF}\text{-}\lambda^{\neg\wedge\exists}$.

We can consider other styles for existential types. For example, more implicit type assignment system for existential types was introduced in [15], whose typing rules for existential types are

$$\frac{\Gamma \vdash N : A[X := B]}{\Gamma \vdash \langle \exists, N\rangle : \exists X.A}\ (\exists\mathsf{I}), \qquad \frac{\Gamma_1 \vdash M : \exists X.A \quad \Gamma_2, x : A \vdash N : C}{\Gamma_1, \Gamma_2 \vdash M[x.N] : C}\ (\exists\mathsf{E}).$$

Our proof of the reduction from TC to TI cannot be adapted to this variant, because the term $J_{M,A}$ explicitly contains the information of type A, whereas terms in the variant contain no explicit type information. Moreover, undecidability of TC and TI in the variant has not been known yet. So it would be future work to study the relationship between TC and TI in the variant, and decidability of them.

References

1. G. Barthe and M.H. Sørensen, Domain-free pure type systems. *Journal of Functional Programming* 10:412–452, 2000.
2. O. Danvy and A. Fillinski, Representing Control: a Study of the CPS Translation. *Mathematical Structures in Computer Science* 2(4):361–391, 1992.
3. K. Fujita, Explicitly typed $\lambda\mu$-calculus for polymorphism and call-by-value. In *Proceedings of 4th International Conference on Typed Lambda Calculi and Applications (TLCA 1999)*, LNCS 1581, pp. 162–177, 1999.
4. K. Fujita and A. Schubert, Partially Typed Terms between Church-Style and Curry-Style. In *International Conference IFIP TCS 2000*, LNCS 1872, pp. 505–520, 2000.
5. K. Fujita, Galois embedding from polymorphic types in to existential types. In *Proceedings of 7th International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, LNCS 3461, pp. 194–208, 2005.
6. R. Harper and M. Lillibridge, Polymorphic Type Assignment and CPS Conversion. *Lisp and Symbolic Computation*, 6:361–380, 1993.
7. M. Hasegawa, Relational parametricity and control. *Logical Methods in Computer Science*, 2(3:3):1–22, 2006.
8. M. Hasegawa, Unpublished manuscript, 2007.
9. A.J. Kfoury, J. Tiuryn and P. Urzyczyn, The Undecidability of the Semi-unification Problem. *Information and Computation* 102:83–101, 1993.
10. J.C. Mitchell and G.D. Plotkin, Abstract types have existential type. *ACM Transactions on Programming Languages and Systems* 10(3):470–502, 1988.
11. K. Nakazawa, M. Tatsuta, Y. Kameyama, and H. Nakano, Undecidability of Type-Checking in Domain-Free Typed Lambda-Calculi with Existence. In *the 17th EACSL Annual Conference on Computer Science Logic (CSL 2008)*, LNCS 5213, pp. 477–491, 2008.
12. K. Nakazawa and M. Tatsuta, Type Checking and Inference for Polymorphic and Existential Types. In *the 15th Computing: the Australasian Theory Symposium (CATS 2009)*, Conferences in Research and Practice in Information Technology (CRPIT), Vol. 94, 2009.
13. M. Parigot, $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LNCS 624, pp.190–201, 1992.
14. A. Schubert, Second-order unification and type inference for Church-style polymorphism. In *the 25th Annual ACM Symposium on Principles of Programming Languages (POPL '98)*, pp.279–288, 1998.
15. M. Tatsuta, Simple saturated sets for disjunction and second-order existential quantification. In *Proceedings of 8th International Conference on Typed Lambda Calculi and Applications (TLCA 2007)*, LNCS 4583, pp. 366–380, 2007.
16. M. Tatsuta, K. Fujita, R. Hasegawa, and H. Nakano, Inhabitance of Existential Types is Decidable in Negation-Product Fragment. In *Proceedings of 2nd International Workshop on Classical Logic and Computation (CLC2008)*, 2008.

17. H. Thielecke, Categorical Structure of Continuation Passing Style. Ph.D. Thesis, University of Edinburgh, 1997.
18. J.B. Wells, 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 '94)*, pp. 176–185, 1994.

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

Manuel Hernández

Universidad Tecnológica de la Mixteca, Huajuapan de Lon, 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 structuring knowledge for computer programming [Par76,Dar78,Lau89,WZ96]. Some advantages of taxonomies of algorithms are: a) We place each algorithm in parent-child relationships; b) we uncover the rationale behind the design of each algorithm; and, c) we can know better how an algorithm works, increasing our comprehension of how an algorithm can efficiently be implemented or automatically derived. A taxonomy can also reveal certain faults in the design of an algorithm. We can even discover minor variants as by-products of our main developments. We will show program transformation is not only a 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 and decreasing of nondeterminism, 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 algorithm [KMP77] belongs to the first group. The search part of the Knuth–Morris–Pratt algorithm has been the subject of intensive study in program derivation [CD89,Smi91,SGJ96]. However, the Knuth–Morris–Pratt algorithm is hardly used in practice. Instead, string searching is often implemented following the Boyer–Moore algorithm [Ste94] or some variant; these algorithms belong to the second group. In this work we derive the complete version of the search part of the Boyer–Moore algorithm and some of its variants using logic program transformation. By using the powerset construction, deterministic unfolding, constraint introduction via the clause splitting rule, and extended folding, we show how to obtain Boyer–Moore-like programs from naive and nondeterministic logic programs.

Our main interest is centered on the search part of each algorithm, so that we want to maximize the shift values and to minimize the number of comparisons given a mismatch between the pattern and a portion of the text. The algorithms to discuss are: The Boyer–Moore algorithm itself (BM algorithm, for short); the Horspool variant (BMH algorithm); the Partsch–Stomp variant (BMPS algorithm); and, the Apostolico–Giancarlo variant (BMAG algorithm). Some minor variants will be byproducts of our main developments.

These algorithms are derived for the following reasons. The BM algorithm is one of the most valuable algorithms for string-matching in practice. The BMH variant is highly practical for ordinary alphabets, such as the English alphabet. Its cost of preprocessing is low, and it is a simplification of the BM algorithm. The BMPS algorithm surpasses the BM algorithm in the search stage (the shifts given by the BM are minor than those of the BMPS variant), although the cost of preprocessing is high. Finally, the BMH and the BMPS variants, and the BM itself, have the deficiency of comparing again some characters already previously treated, but in the BMAG variant these re-comparisons are avoided because this variant incorporates in its design a *memory*. The method to place an algorithm in the taxonomic tree is: A naive logic program solving the string-matching problem is presented, a derivation is carried out, and then the derivation is associated with a path, from the root to the leaf, of the taxonomic tree given in Fig. 1 (where by *pre-algorithms* we mean some intermediate programs within the derivation). Leaves indicate a complete and terminal development of a derivation.

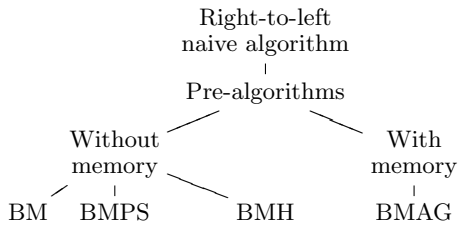


Fig. 1: A taxonomic tree of some right-to-left string-matching algorithms.

1.1 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 Boyer–Moore algorithm. Section 3 gives some basic 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 some related work and in Section 6 are given 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*. The set $\mathcal{A}^*$ consists of finite sequences of characters of $\mathcal{A}$ and these sequences are called *strings*. We denote by $\mathrm{Set}(S)$ the set of elements ocurring 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. We define the concatenation of S and T, $S +\!+ T$, as 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. 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 $S_2 = U +\!+ S_1 +\!+ V$.

Let P and T be strings. 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 configurations is to explain how a new configuration is reached from a previous one, given a rule to do this transition. A *matching schedule* is an order (or permutation) to make comparisons between characters of P and those in a window of T when trying to find P in T.

Naive string-matching algorithms. A first straightforward 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. 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. This is sometimes called the *naive left-to-right string-matching algorithm*. A second straightforward algorithm solving the exact string-matching problem originates the *naive right-to-left string-matching algorithm*: this time 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]. This situation, however, is unlikely to occur in practice, and 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 named 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.

By inspecting the text character aligned with the rightmost pattern character, we have two cases: In case such a text character does not occur in the pattern, we can slide the pattern *its whole length*, to surpass the bad character of the text; as a second case, if the text character is also in the pattern we can shift the pattern a number of characters determined by the preprocessing phase, trying to align these good characters. If a partial match is discovered, in the next configuration those rightmost parts that match would be aligned again, so that transitions are ruled by the longest partial matches.

In case of a mismatch this algorithm uses two precomputed functions to shift the pattern to the right with respect to the text. These two shift 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 number of characters shifted is 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 the 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.

```
initializeBM(P, δ1, δ2);    {preprocessing stage to tabulate the δ1,δ2 functions}
i := m; j := m;
while (j > 0) and (i ≤ 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(δ1(t_i), δ2(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. 2: The Boyer–Moore algorithm.

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. It may also happen that such a reoccurrence may "fall off" the left end of P: 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:

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

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 2 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 as well as 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 $q \leftarrow s \wedge q$, 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*:

$$member(E, [A|Ls]) \leftarrow E = A \tag{9}$$
$$member(E, [A|Ls]) \leftarrow member(E,Ls) \tag{10}$$
$$nonmember(E, Ls]) \leftarrow \backslash + \; member(E,Ls) \tag{11}$$

It is also possible to introduce inequations expressed as inequalities as have been shown in [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:

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

We note that in the following set of clauses each clause shares with other the subgoal $X = a$. We name this common subgoal a *pivot*. The pivot in the following set of clauses is boxed-in:

$$r(X,Y) \leftarrow \boxed{X = a} \wedge Y = a \tag{15}$$
$$r(X,Y) \leftarrow \boxed{X = a} \wedge Y = b \tag{16}$$

For folding, we create a new definition: $g(Y) \leftarrow Y = a$ and $g(Y) \leftarrow Y = b$. Folding with respect to the pivot $X = a$ and the new definition, we have the following 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:

$$\mathit{substring}(P,T) \leftarrow \mathit{append}(X_1,P,X_2) \wedge \mathit{append}(X_2,X_3,T) \tag{17}$$

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: $A_i = B_i \wedge A_{i+1} = B_{i+1} \wedge \ldots \wedge A_j = B_j$. 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 (17), and by using some standard techniques of logic program transformation and partial deduction, we obtain the following clauses:

Program 1

$$ss_P([p_1 : p_m \,|\, L]) \leftarrow \tag{18a}$$
$$ss_P([A \,|\, L]) \leftarrow ss_P(L) \tag{18b}$$

This nondeterministic program can be read as a direct implementation of the left-to-right string-matching algorithm, following the traditional unification algorithm over terms. However, to explicitly control the order of matching via an equational theory, we introduce equations in the body of clause (18a), to get:

Program 2

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_1 : A_m = p_1 : p_m \tag{19a}$$
$$ss_P([A \,|\, L]) \leftarrow ss_P(L) \tag{19b}$$

To deal with a matching schedule from right to left, we invert the order of the equations of the clause (19a), and because we use a set of variables $A_1, \ldots, A_m$ to access elements of the pattern, we apply a substitution to clause (19b):

Program 3

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{20a}$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow ss_P([A_2 : A_m \,|\, L]) \tag{20b}$$

This program disallows to use texts of length lesser than m. Because this matching schedule, each particular mismatched character involves a suffix of the pattern (included 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 4

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{21a}$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m \neq p_m \wedge ss_P([A_2 : A_m \,|\, L]) \tag{21b}$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m = p_m \wedge A_{m-1} \neq p_{m-1} \wedge ss_P([A_2 : A_m \,|\, L]) \tag{21c}$$
$$\vdots$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_2 = p_m : p_2 \wedge A_1 \neq p_1 \wedge ss_P([A_2 : A_m \,|\, L]) \tag{21d}$$
$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_1 = p_m : p_1 \wedge ss_P([A_2 : A_m \,|\, L]) \tag{21e}$$

We note that each clause in Prog. 4 corresponds with one of the following cases: Clauses (21a) and (21e) deal with a total match and possible reoccurrences of the pattern, respectively. Clause (21b) treats an initial mismatching. The other clauses find partial matchings:

$$ss_P([A_1 : A_m \,|\, L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \wedge ss_P([A_2 : A_m \,|\, L]) \tag{22}$$

where k is such that $1 \leq k \leq m-1$. Each clause is unfolded enough with respect to $ss_P/1$ through a succession of deterministic unfolding steps.

In Prog. 4 the nondeterminism has been increased, but we can do 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 5

$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m = p_m \wedge new_1([A_1 : A_m \mid L]) \tag{23}$$
$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m \wedge ss_P([A_2 : A_m \mid L]) \tag{24}$$

$$new_1([A_1 : A_m \mid L]) \leftarrow A_{m-1} = p_{m-1} \wedge new_2([A_1 : A_m \mid L]) \tag{25}$$
$$new_1([A_1 : A_m \mid L]) \leftarrow A_{m-1} \neq p_{m-1} \wedge ss_P([A_2 : A_m \mid L]) \tag{26}$$
$$\ldots$$
$$new_{m-1}([A_1 : A_m \mid L]) \leftarrow A_1 = p_1 \wedge new_m([A_1 : A_m \mid L]) \tag{27}$$
$$new_{m-1}([A_1 : A_m \mid L]) \leftarrow A_1 \neq p_1 \wedge ss_P([A_2 : A_m \mid L]) \tag{28}$$

$$new_m([A_1 : A_m \mid L]) \leftarrow \tag{29}$$
$$new_m([A_1 : A_m \mid L]) \leftarrow ss_P([A_2 : A_m \mid L]) \tag{30}$$

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 $[A_2 : A_m \mid L]$ 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:

$$shift(1, [A \mid L_1], L_1) \leftarrow \tag{31a}$$
$$shift(N, [A \mid L_1], L_2) \leftarrow N_1 \; is \; N - 1 \wedge shift(N_1, L_1, L_2) \tag{31b}$$

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 Clause (20b):

Program 6

$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{32a}$$
$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m = p_m \wedge ss_P([A_2 : A_m \mid L]) \tag{32b}$$
$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m \wedge ss_P([A_2 : A_m \mid L]) \tag{32c}$$

We notice that the negative information of $A_k \neq p_k$ in Clause (32c) 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 (32c), using $A_m \in \mathcal{S}_p \vee A_m \notin \mathrm{Set}(P)$, to obtain the following other two clauses:

$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m \wedge A_m \in \mathrm{Set}(P) \wedge ss_P([A_2 : A_m \mid L]) \tag{33a}$$
$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m \wedge A_m \notin \mathrm{Set}(P) \wedge ss_P([A_2 : A_m \mid L]) \tag{33b}$$

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 (33a) we get the following new clauses:

$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m \wedge A_m = p_1 \wedge ss_P([A_2 : A_m \mid L]) \tag{34}$$
$$\vdots$$
$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m \wedge A_{m-1} = p_{m-1} \wedge ss_P([A_2 : A_m \mid L]) \tag{35}$$

whereas from the clause (33b) we get the clause:

$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m \neq p_m \wedge A_m \neq p_1 \wedge \ldots \wedge A_m \neq p_m \wedge ss_P([A_2 : A_m \mid L]) \tag{36}$$

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 (32b)), we have a correspondent shift value. These values are asserted as facts, and these facts tabulate the δ_1 function:

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

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

Program 7

$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m : A_1 = p_m : p_1 \tag{37a}$$
$$ss_P([A_1 : A_m \mid L]) \leftarrow d1(A_m, Val) \wedge shift(Val, [A_2 : A_m \mid L], L_1) \wedge ss_P(L_1) \tag{37b}$$

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 (32b), according to Boyer and Moore, but deterministic unfolding could be applied without any problem to (32b).

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 $[A_2 : A_n \mid L]$ 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 $ss_P/1$ in Prog. 3.

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:

$$ss_P([A_1 : A_m \mid L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \wedge ss_P([A_2 : A_m \mid L]) \quad (38)$$

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:

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

Unfolding each clause of Prog. 4 we would get the δ_2 function [HR01]. By using the *shift*/3 function:

$$\begin{aligned} ss_P([A_1 : A_m \mid L]) \leftarrow\ & d2(ssuf(k), Vs_k) \wedge \\ & shift(Vs_k, [A_2 : A_m \mid L], L_1)) \wedge ss_P(L_1) \end{aligned} \quad (39)$$

When we execute Prog. 5, 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 (21e), 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:

$$\begin{aligned} ss_P([A_1 : A_m \mid L]) \leftarrow\ & d1(c, Val_1) \wedge d2(ssuf(k), Val_2) \wedge \\ & \max(Val_1 - (m-k), Val_2, Val) \wedge \\ & shift(Val, [A_1 : A_m \mid L], L_1) \wedge ss_P(L_1) \end{aligned} \quad (40)$$

($Val_1 - (m-k)$ 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

$$ss_P([A_1 : A_n \,|\, L]) \leftarrow A_m = p_m \wedge ss_P([A_2 : A_n \,|\, L]) \tag{41}$$

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 (33a), 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 $p_s \in \mathrm{Set}(P)$. 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 most 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]. In the next paragraph we will see an algorithm endowed with a memory, the BMAG algorithm.

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 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 (21e) 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

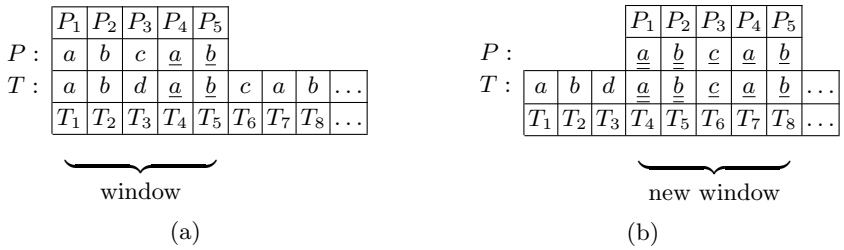


Fig. 3: Configurations and a transition in the BMAG algorithm.

the configuration given in (a), Fig. 3, where a mismatch occurs 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. 3, 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:

$$ss_P([A_1 : A_m|L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \wedge \tag{42}$$

$$\mathit{shift}(d(k), [A_2 : A_m|L], L_1) \wedge ss_P(L_1) \tag{43}$$

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.

$$ss_P^{p_m : p_{m-d(k)}} p([A_1 : A_m|L]) \leftarrow A_m : A_{m-d(k)} = p_m : p_{m-d(k)} \tag{44}$$

Hence, $d(k)$ characters are not more revisited. The new formulation of (43) is:

$$ss_P([A_1 : A_m|L]) \leftarrow A_m : A_{k+1} = p_m : p_{k+1} \wedge A_k \neq p_k \wedge \tag{45}$$

$$\mathit{shift}(d(k), [A_2 : A_m|L], L_1) \wedge ss_P^{p_m : p_{m-d(k)}}(L_1) \tag{46}$$

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 to $ss_P/1$ (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:

$$\mathit{new}([B_1 : B_m \,|\, L]) \leftarrow B_m : B_{m-k} = p_m : p_{m-k} \tag{47}$$

$$\mathit{new}([B_1 : B_m \,|\, L]) \leftarrow ss_P([B_1 : B_m|L]) \tag{48}$$

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 kind 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 to obtain specific algorithms some human assistance is necessary. 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 variants of an algorithm via logic program transformation. The selection of design decisions has been guided by known algorithms. 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 showed at least for the BM variant restricted to use only the δ_2 function in [HR03].

As future work, a proposal is to add some other 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], and that hybrid algorithm described in [CP91], based on a complex preprocessing by factoring of the pattern.

Other possible taxonomies related to text processing would be based on taking the text as static, instead of the pattern. McCreight and Ukkonen algorithms would be good candidates to be derived. Approximate string-matching is also another area to be explored.

Acknowledgments

Support for this work came from *Consejo Nacional de Tecnología* (CONACYT) and *Universidad Tecnológica de la Mixteca* (UTM). I would like to thank David Rosenblueth by sharing with me some of his views on program transformation. I also grateful the reviewers of WFLP'09 for their valuable comments.

References

[AG86] Alberto Apostolico and Rafaelle Giancarlo. The Boyer–Moore–Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, February 1986.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[Bir05] Richard Bird. Polymorphic string matching. In *Haskell'05*, Tallin,Estonia, September 2005. ACM.

[BM77] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Communications of the Association for Computing Machinery*, 20(10), October 1977.

[CD78] Keith L. Clark and John Darlington. Algorithm classification through synthesis. *The computer journal*, 23(1):61–65, 1978.

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

[Cla78] Keith L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[CP91] Maxime Crochemore and Dominique Perrin. Two-way string-matching. *Journal of the ACM*, 38(3), July 1991.

[CR94] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.

[CR02] Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific Publishing, 2002.

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

[DRK06] Olivier Danvy and Henning Rohde Korsholm. On obtaining the boyer–moore string-matching algorithm by partial evaluation. *Information Processing Letters*, 99(4), August 2006.

[FPP02] Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Specialization with clause splitting for deriving deterministic constraint logic programs. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC'02)*, Hammamet,Tunisia, October 2002. IEEE Computer Society Press.

[GK94] Manolis Gergatsoulis and Maria Katzouraki. Unfold/fold Transformations for Definite Clause Programs. In Manuel Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*. 6th International Symposium, PLILP'94, Springer-Verlag, September 1994.

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

[HR01] Manuel Hernández and David Rosenblueth. 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'01)*, pages 38–48, Florence, Italy, September 2001.

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

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal of Computation*, 6(2):323–350, June 1977.

[Lau89] Kung-Kiu Lau. A Note on Synthesis and Classification of Sorting Algorithms. *Acta Informatica*, 27:73–80, 1989.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[MACD01] Karoline Malmkjr, Torben Amtoft, Charles Consel, and Olivier Danvy. The abstraction and instantiation of string-matching programs. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, number 2566 in Lecture Notes in Computer Science*, pages 332–357. Springer-Verlag, 2001.

[Par76] David L. Parnas. On the design and development of program families. *IEEE Transactions of Software Engineering*, SE-2(1):1–9, March 1976.

[Pep91] Peter Pepper. Literate Program Derivation: A Case Study. In M. Broy and M. Wirsing, editors, *Methods of Programming*, volume 544 of *Lecture Notes on Computer Science*, pages 101–124. Springer-Verlag, 1991.

[PP98] Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.

[PPR97] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. Enhancing Partial Deduction via Unfold/Fold Rules. In *Logic Program Synthesis and Transformation*, volume 1207 of *Lecture Notes in Computer Science*. LOPSTR'96, Springer-Verlag, August 1997.

[PS90] Helmut A. Partsch and Frank A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, 2:109–122, 1990.

[RS83] John H. Reif and William L. Scherlis. Deriving efficient graphs algorithms. In Springer-Verlag, editor, *Logic of programs (Proceedings 1983)*, volume 164, pages 421–441. LNCS, 1983.

[SGJ96] Morten H. Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of functional programming*, 6(6):811–838, 1996.

[Smi91] Donald A. Smith. Partial Evaluation of Pattern Matching in Constraint Logic Programming Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation,PEPM'91*, pages 62–71, Connecticut, USA, June 1991. ACM Press.

[Ste94] Graham A. Stephen. *String Searching Algorithms*, volume 3 of *Lecture Notes Series on Computing*. World Scientific Publishing, 1994.

[Sun90] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8), August 1990.

[WZ96] Bruce W. Watson and Gerard Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. *Science of Computer Programming*, 27(2):85–118, September 1996.

A Simple Region Inference Algorithm for a First-Order Functional Language *

Manuel Montenegro, Ricardo Peña, 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 [8, 7]. 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 navigators.

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. In *Safe*, regions are allocated/deallocated by following a stack discipline associated to 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 [9]. 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.

$$
\begin{array}{lll}
prog \rightarrow \overline{data_i}^n ; \overline{dec_j}^m ; e & & \\
data \rightarrow \textbf{data}\ T\ \overline{\alpha_i}^n\ @\ \overline{\rho_j}^m = \overline{C_k\ \overline{t_{ks}}^{n_k}\ @\ \rho_m}^l & & \{\text{recursive, polymorphic data type}\} \\
dec \rightarrow f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e & & \{\text{recursive, polymorphic function}\} \\
e \rightarrow a & & \{\text{atom: literal } c \text{ or variable } x\} \\
\quad \mid f\ \overline{a_i}^n\ @\ \overline{r_j}^l & & \{\text{function application}\} \\
\quad \mid C\ \overline{a_i}^n\ @\ r & & \{\text{constructor application}\} \\
\quad \mid \ldots & & \text{let, case } \ldots
\end{array}
$$

Fig. 1. Simplified *Safe*

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*. If polymorphic recursion is not inferred, it runs in time $O(n)$ in the worst case, being n as above, while if polymorphic recursion appears, it needs time $O(n^2)$ 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 region annotations. 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. In general we allow pattern matching in function definitions and also

$$\frac{(f\ \overline{x_i}^n @\ \overline{r_j}^m = e) \in \Sigma \quad [\overline{x_i \mapsto E(a_i)}^n, \overline{r_j \mapsto E(r'_j)}^m, self \mapsto k+1] \vdash h, k+1, e \Downarrow h', k+1, v}{E \vdash h, k, f\ \overline{a_i}^n @\ \overline{r'_j}^m \Downarrow h' \mid_k, k, v}\ [App]$$

$$\frac{j \leq k \quad fresh(p)}{E[r \mapsto j, \overline{a_i \mapsto v_i}^n] \vdash h, k, C\ \overline{a_i}^n @ r \Downarrow h \uplus [p \mapsto (j, C\ \overline{v_i}^n)], k, p}\ [Cons]$$

Fig. 2. Operational semantics of *Safe* expressions

expressions in function applications, and we use them in the examples of this paper. However, in order to simplify the presentation of the operational semantics and the typing rules we will omit them there.

Region annotations are region type variables $\rho, \rho_i \ldots$ in datatype definitions and types, and region variables $r, r_i \ldots$ in function definitions and applications, and in constructor applications.

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 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 separate DS. 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:

$$\textbf{data}\ \mathit{Table}\ a\ b\ @\ \rho_1\ \rho_2\ \rho_3 = \mathit{TBL}\ ([(a,b)@\ \rho_1]@\ \rho_2)@\ \rho_3$$

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 $\mathit{concat} :: \forall a\rho_1\rho_2.[a]@\rho_1 \to [a]@\rho_2 \to \rho_2 \to [a]@\rho_2$ 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 $\mathit{partition} :: \forall\rho_1\rho_2\rho_3\rho_4.\mathit{Int} \to [\mathit{Int}]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([\mathit{Int}]@\rho_2, [\mathit{Int}]@\rho_3)@\rho_4$. 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:

$$\begin{aligned}
\mathit{makeTree} &:: \forall a\rho_1\rho_2.[a]@\rho_1 \to \rho_2 \to \mathit{Tree}\ a@\rho_2\\
\mathit{inorder} &:: \forall a\rho_1\rho_2.\mathit{Tree}\ a@\rho_1 \to \rho_2 \to [a]@\rho_2\\
\mathit{treesort} &:: \forall a\rho_1\rho_2.[a]@\rho_1 \to \rho_2 \to [a]@\rho_2
\end{aligned}$$

After region inference, the definition is annotated as follows:

$$\frac{\begin{array}{c}\text{fresh}(\rho_{self}),\quad \rho_{self} \notin \mathit{regions}(s)\quad \mathcal{R} = \mathit{regions}(\overline{t_i}^{\,n}) \cup \{\overline{\rho_j}^{\,l}\} \cup \mathit{regions}(s) \\ \Gamma + \overline{[x_i : t_i]}^{\,n} + \overline{[r_j : \rho_j]}^{\,l} + [\mathit{self} : \rho_{self}] + [f : \forall \rho \in \mathcal{R} \,.\, \overline{t_i}^{\,n} \to \overline{\rho_j}^{\,l} \to s] \vdash e : s\end{array}}{\{\Gamma\}\ \ f\ \overline{x_i}^{\,n}\ @\ \overline{r_j}^{\,l} = e\ \ \{\Gamma + [f : \mathit{gen}(\forall \rho \in \mathcal{R} \,.\, \overline{t_i}^{\,n} \to \overline{\rho_j}^{\,l} \to s, \Gamma)]\}}\ \text{[FUNB]}$$

$$\frac{\Sigma(C) = \sigma \quad \overline{s_i}^{\,n} \to \rho \to T\ @\overline{\rho}^{\,m} \trianglelefteq \sigma \quad \Gamma = (\overline{[a_i : s_i]}_{i=1}^{n}) + [r : \rho]}{\Gamma \vdash C\ \overline{a_i}^{\,n} @ r : T\ @\overline{\rho}^{\,m}}\ \text{[CONS]}$$

Fig. 3. Rule for function definitions

```
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.

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. 3.

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 $\mathit{gen}(\sigma', \Gamma)$ 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.

```
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. 4. A high-level view of the Hindley-Milner inference algorithm

3.1 A high-level view of the algorithm

Figure 4 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. Finally, the main expression `exp` of the program is inferred, and decorated by `decorAndGenMainExp` (not shown).

3.2 Region inference of function definitions

Figure 5 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. 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

$$
\begin{array}{l}
\mathit{decorAndGenOuterDefs}\ \Gamma\ \mathit{Defs} = (\Gamma \cup [f \mapsto t^+], \mathit{Defs}'') \\
\quad \textbf{where}\ f \qquad\qquad\qquad\qquad\qquad\quad = \mathit{extractFunctionName}\ \mathit{Defs} \\
\qquad\quad (\mathit{Defs}', \mathit{Eqs}, \mathit{Fresh}_{expl}, \overline{trec_j}^{\,p}) = \mathit{decorAndGenEqs}\ \Gamma\ \mathit{Defs} \\
\qquad\quad \theta_1 \qquad\qquad\qquad\qquad\qquad\quad = \mathit{solveEqs}\ \mathit{Eqs} \\
\qquad\quad t \qquad\qquad\qquad\qquad\qquad\qquad\quad = \theta_1(\mathit{type}\ \mathit{Defs}') \\
\qquad\quad (\theta_2, \overline{\varphi_j}^{\,p}) \qquad\qquad\quad = \mathit{handleRecCalls}\ t\ (\theta_1(\overline{trec_j}^{\,p})) \\
\qquad\quad \theta \qquad\qquad\qquad\qquad\qquad\qquad\quad = \theta_2 \circ \theta_1 \\
\qquad\quad R_{expl} \qquad\qquad\qquad\qquad\qquad\quad = \theta(\mathit{Fresh}_{expl}) \\
\qquad\quad (\theta_{self}, t^+, \mathit{RegMap}) \qquad = \mathit{inferRegions}\ t\ R_{expl}\ \overline{\varphi_j}^{\,p} \\
\qquad\quad \mathit{Defs}'' \qquad\qquad\qquad\qquad\qquad\quad = \mathit{annotateDef}\ (\theta_{self} \circ \theta)\ \mathit{RegMap}
\end{array}
$$

Fig. 5. HM-type and region inference for a single function

$$
\begin{array}{l}
\mathit{inferRegions}\ t\ R_{expl}\ \overline{\varphi_j}^{\,p} = ([\rho \mapsto \rho_{self} \mid \rho \in R_{self}], \overline{t_i}^{\,n} \to \overline{\rho_k}^{\,m} \to t', [\overline{\rho_k \mapsto r_j}^{\,m}]) \\
\quad \textbf{where}\ \overline{t_i}^{\,n} \to t' \qquad = t \\
\qquad\quad R_{out} \qquad\qquad = \mathit{regions}\ t' \\
\qquad\quad R_{in} \qquad\qquad = \mathit{regions}\ \overline{t_i}^{\,n} \\
\qquad\quad R_{arg} \qquad\qquad = R_{expl} \cap (R_{in} \cup R_{out}) \\
\qquad\quad (R'_{arg}, R'_{expl}) = \mathit{computeRargFP}\ R_{in}\ R_{out}\ R_{arg}\ R_{expl}\ \overline{\varphi_j}^{\,p} \\
\qquad\quad R_{self} \qquad\qquad = R'_{expl} - (R_{out} \cup R_{in}) \\
\qquad\quad \overline{\rho_k}^{\,m} \qquad\qquad = R'_{arg}
\end{array}
$$

$$
\begin{array}{l}
\mathit{computeRargFP}\ R_{in}\ R_{out}\ R_{arg}\ R_{expl}\ \overline{\varphi_j}^{\,p} \\
\qquad\quad \mid R_{arg} == R'_{arg} = (R'_{arg}, R'_{expl}) \\
\qquad\quad \mid \textbf{otherwise} \quad = \mathit{computeRargFP}\ R_{in}\ R_{out}\ R'_{arg}\ R'_{expl}\ \overline{\varphi_j}^{\,p} \\
\quad \textbf{where}\ R'_{expl} = R_{expl} \cup \bigcup_{j=1}^{p} \{\varphi_j(\rho) \mid \rho \in R_{arg}\} \\
\qquad\quad R'_{arg} \ = R'_{expl} \cap (R_{in} \cup R_{out})
\end{array}
$$

Fig. 6. Second phase of the region inference algorithm

substitution φ_j reflects this fact by mapping the region type variables in t to those in $trec_j$.

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. 6 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* 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. 6 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 to 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 Appendix **??** 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. 6) 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, i.e. maximises garbage destruction.

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.

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 $Fresh_{expl} = \{\rho_2, \ldots, \rho_8\}$. We show them as annotations in the program just in order to better explain the example:

$$\begin{array}{ll} partition\ y\ [\,] = ([\,] :: \rho_2, [\,] :: \rho_3) :: \rho_4 & \\ partition\ y\ (x : xs) & \mid x \leq y = (x : ls :: \rho_5, gs) :: \rho_6 \\ & \mid x > y = (ls, x : gs :: \rho_7) :: \rho_8 \\ & \textbf{where}(ls, gs) = partition\ y\ xs \end{array}$$

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 $Int \to [Int]@\rho_1 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4$, 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.

$$\begin{array}{rl} \text{Definition: } & Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4 \\ \text{Rec. call: } & Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_9 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_9 \end{array}$$

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:

$$\begin{array}{ll} partition :: Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4 & \\ partition\ y\ [\,]\ @\ r_2\ r_3\ r_4 = ([\,]@r_2, [\,]@r_3)@r_4 & \\ partition\ y\ (x : xs)\ @\ r_2\ r_3\ r_4 & \mid x \leq y = ((x : ls)@r_2, gs)@r_4 \\ & \mid x > y = (ls, (x : gs)@r_3)@r_4 \\ & \textbf{where}\ (ls, gs) = partition\ y\ xs\ @\ r_2\ r_3\ self \end{array}$$

Notice that the tuple resulting from the recursive call to *partition* is located in the working region. Without 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:

$$\begin{array}{l} sumList\ (x : [\,]) = (x : [\,] :: \rho_2) :: \rho_3 \\ sumList\ (x : xs) = (x + y : sumList\ xs) :: \rho_4 \quad \textbf{where}\ (y : \_) = xs \end{array}$$

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 $[Int]@\rho_1$. 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 $[Int]@\rho_1 \to [Int]@\rho_2$. Hence we get $R_{in} = \{\rho_1\}$, $R_{out} = \{\rho_2\}$ and $R_{arg} = \{\rho_2\}$. We extend the signature of *sumList* to $[Int]@\rho_1 \to \rho_2 \to [Int]@\rho_2$.

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:

$$
\begin{array}{l}
sumList \quad :: [Int]@\rho_1 \to \rho_2 \to [Int]@\rho_2 \\
sumList\ (x : [\,])\ @\ r = (x : ([\,] \ @\ r))@\ r \\
sumList\ (x : xs)\ @\ r = (x + y : sumList\ xs\ @\ r)\ @\ r \quad \textbf{where}\ (y : \_) = xs
\end{array}
$$

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:

$$
\begin{array}{l}
pascal\ 0 = (1 : [\,] :: \rho_1) :: \rho_1 \\
pascal\ n = (1 : sumList\ (pascal\ (n-1))) :: \rho_1
\end{array}
$$

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:

$$
\begin{array}{l}
pascal \quad :: Int \to \rho_1 \to [Int]@\rho_1 \\
pascal\ 0\ @\ r = (1 : [\,] \ @\ r)@\ r \\
pascal\ n\ @\ r = (1 : sumList\ (pascal\ (n-1)\ @\ self)\ @\ r)\ @\ r
\end{array}
$$

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 polymorphic recursion the result of every recursive call would be built in the output region r, which would imply a cost of $O(n^2)$ in space.

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 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 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. Alloca-

tion 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.

References

1. A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation, PLDI'95*, pages 174–185. ACM Press, 1995.
2. L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *23rd ACM Symposium on Principles of Programming Languages, POPL'96*, pages 171–183. ACM Press, 1996.
3. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation, PLDI'02*, pages 282–293, 2002.
4. F. Henglein, H. Makholm, and H. Niss. 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'01*, pages 175–186. ACM Press, 2001.
5. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *30th ACM Symposium on Principles of Programming Languages, POPL'03*, pages 185–197. ACM Press, 2003.
6. H. Makholm. A language-independent framework for region inference. Ph.D thesis, Univ. of Copenhagen, Dep. of Computer Science, Denmark, 2003.
7. M. Montenegro, R. Peña, and C. Segura. 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'08*, pages 152–162, 2008.
8. M. Montenegro, R. Peña, and C. Segura. An inference algorithm for guaranteeing safe destruction. In *18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08, Lecture Notes in Computer Science 5438*, pages 135–151, 2009.
9. R. Peña and C. Segura. Formally deriving a compiler for SAFE. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL'06, Technical Report, 2006-S01. Eötvös Loránd University*, pages 429–446, 2006.
10. M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, 1998.
11. M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark, 2006.
12. M. Tofte and N. Hallenberg. Region-Based Memory Management in Perspective. In *Invited talk Space 2001 Work., London*, pages 1–8. Imperial College, Jan. 2001.
13. M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *21st ACM Symposium on Principles of Programming Languages, POPL'94*, pages 188–201, Jan. 1994.
14. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

pFun: A semi-explicit parallel purely functional language

André R. Du Bois[1], Gerson Cavalheiro[2], Juliana Vizzotto[3]

[1] PPGInf - UCPel dubois@ucpel.tche.br
[2] UFPel
gerson.cavalheiro@ufpel.edu.br
UNIFRA juvizzotto@gmail.com

Abstract. In this paper we present the design and implementation of *p*Fun, a semi-explicit parallel purely functional language. Parallelism is introduced in *p*Fun through annotations. These annotations indicate expressions that can be evaluated in parallel with the rest of the program. Task creation, synchronization and scheduling of computations on processors are managed automatically by *p*Fun's runtime system. *p*Fun's programming model and runtime system are described, and preliminary measurements of the current prototype implementation on an SMP-machine, a Beowulf Cluster and an small heterogeneous GRID are presented.

1 Introduction

With the arrival of multi-core chips for domestic computers, parallel programming is becoming a mandatory feature in programming languages. Furthermore, networks are increasingly pervasive, and big companies are using clusters or grids to solve large-scale computation problems. Parallel programming is hard, and it is difficult to find programmers that fully understand concurrency [23], even more considering the diversity of parallel architectures.

To address these challenges, we advocate the need of a programming platform that hides most aspects of the architecture being used and limits the programmer's task to annotate expressions in the program that may be evaluated in parallel. Such a programming platform must be supported by an advanced runtime system that provides automatic creation, synchronization and scheduling of computations on processors.

Purely functional languages have an interesting property, called *referential transparency*, that allows subexpressions of a purely functional program to be evaluated in parallel in any order, always delivering the same final result. Subexpressions of a program will never have implicit control dependencies between them, as the ones introduced by assignment [11]. Although referential transparency allows the evaluation of any sub-expression of a program in parallel, in practice that would generate tasks that are too small in comparison with the overhead of creating concurrent tasks to evaluate them [13]. Semi-explicit parallel functional languages [11] are languages that provide a mechanism to annotate

potentially parallel tasks and to control the granularity of these tasks but hide from the programmer aspects as task creation, synchronization and scheduling.

In this paper, we present *p*Fun, a strict, strongly typed, semi-explicit parallel functional language. Parallelism is introduced in *p*Fun through two constructs: `par` and `sync`, that provide an abstraction similar to Multilisp's *futures* [20] (Section 2.1). Creation, distribution and synchronization of tasks are left to the implementation of the language, i.e., its runtime system. The `par` and `sync` primitives are low level constructs to express parallelism and they can be used to implement higher-level coordination primitives, such as algorithmic skeletons, i.e., higher-order functions that encapsulate common patterns of parallel computing [6].

The execution of programs on heterogeneous environments is provided by compiling programs into architecture-independent byte-code, and *p*Fun's runtime system provides ways for serializing and communicating computations between different processes or hosts. *p*Fun's runtime system is implemented using standard C and TCP/IP sockets for communication, maintaining a high degree of portability. Distribution and scheduling of tasks is provided automatically by the runtime system that is based on the GUM [25] virtual machine (Section 3.1).

This paper describes the design and implementation of *p*Fun. The paper is organized as follows: In the next section, we explain the *p*Fun language and its primitives for parallel programming through examples. In section 3, *p*Fun's runtime system is described. Section 4 gives preliminary performance measurements of six different parallel programs. Finally, conclusions and future work are discussed in Section 6.

2 The *p*Fun Language

*p*Fun is a *strict* parallel purely functional language with a syntax similar to the Haskell language [17]. The reader should be aware that *p*Fun *is not Haskell*. Its syntax is similar to Haskell only for convenience.

*p*Fun's syntax, semantics and parallel primitives will be presented through examples in the following Sections.

2.1 The *p*Fun primitives for expressing parallelism

The *p*Fun language provides two basic primitives for expressing parallelism: `par` and `sync`. The `par` primitive is used to express potential parallelism. It takes as an argument a *p*Fun expression of any type and returns a reference to a `Par` value that represents an expression that *could* be evaluated in parallel:

```
par :: a -> Par a
```

The `par` primitive only indicates potential parallelism in the program and it does not guarantee that the expression will be evaluated in parallel with the rest of the program. Task creation, scheduling, distribution and synchronization are left to the implementation of the language described in Section 3.

The `sync` primitive receives as an argument a `Par` value and returns the result of the evaluation of that expression:

```
sync :: Par a -> a
```

The operational behavior of the `sync` primitive is to block in its argument if it is being evaluated (by a different processor or remote location) or to create a local thread to evaluate it. The `sync` primitive will only proceed once its argument is evaluated to normal form.

The `par` and `sync` primitives provide the same abstraction as *futures* in Multilisp. The support for futures in Multilisp is given by two primitives, that would have the following types in Haskell:

```
future :: a -> a
touch :: a -> a
```

The `future` primitive creates a task to evaluate its argument and returns a placeholder for the value that will be computed by the task. When an expression needs the value of a future, it can `touch` the future. If the future is unresolved, `touch` will block until the future is completely evaluated. In Multilisp, a future can also be implicitly touched by *strict* operations, i.e., if a strict operation needs the value of a future, it will block until the result is computed. The `future` and `touch` functions have similar expressiveness to `par` and `sync`. In *p*Fun, the only way to get the result of a task is by using `sync`. This resembles the way the future abstraction is provided in modern object oriented languages like Java [9] and PLinQ [7]. Implicit touching of futures in Multilisp adds a high overhead in the implementation as every strict function must check if its argument is a value or a future [20, 21]. As *p*Fun is a strongly typed language checking tags at runtime is not necessary for all operations. The only operation that needs to check tags is `sync`, as described in section 3.4. By wrapping a task in an ADT, *p*Fun's static type system can guarantee that `sync` will always be applied to `Par` objects before they are used.

We plan in the future to investigate how to, based on type information, provide implicit touching of tasks by automatically inserting the `sync` operation in expressions.

2.2 Properties of `par` and `sync`

Semi-explicit parallel functional languages like Multilisp, GpH [24] and *p*Fun break the abstraction offered by functional languages as programmers now have to express *fork/join* parallelism in the code. Although primitives to express fork/join parallelism were introduced, *p*Fun is still a purely functional language. The primitives are well integrated in the language in the sense that the programmer can reason about programs as if they were sequential. For any purely functional expression `exp` written in *p*Fun, the following expression will always evaluate to `true`:

```
sync (par exp) == exp
```

It does not matter if a `Par` value is evaluated locally or on a remote processor, it will always return the same result. Hence, the following function

```
idps :: a -> a
idps x = sync (par x)
```

is equal to the identity function (`id`):

```
id x == idps x
```

for any expression `x` written in *p*Fun.

If someone wants to reason about `par` and `sync`, the `Par` data type should be seen as an ADT:

```
data Par a = Par a

par :: a -> Par a
par x = Par x

sync :: Par a -> a
sync (Par x) = x
```

We are currently developing an operational semantics of *p*Fun so that we can reason not only about the parallel algorithms but also about *coordination*, i.e., how tasks are created, synchronized and distributed. The semantics of *p*Fun is work in progress and a draft version is available [19].

2.3 Example 1: Parallel Fibonacci

In this section we present a naive implementation of a parallel fibonacci function, just to demonstrate the use of the `par` and `sync`. A first parallel version of the fibonacci function can be implemented as:

```
parFib n = if (n<=1) then 1
           else let
                  fib2 = par (parFib (n-2));
                  fib1 = par (parFib (n-1))
                in (sync fib2) + (sync fib1);
```

In the definition of `parFib`, the two recursive calls are *marked* with the `par` primitive to be computed in parallel. It can be very inefficient to create parallel tasks to evaluate every recursive call to `parFib`, since calculating fibonacci of small numbers is a fine grained task. To solve this problem, we can write a more efficient version of `parFib` that only creates parallel tasks when the argument supplied to `parFib` is larger than a threshold:

```
parFib n t = if (n<=t) then (seqFib n)
         else let
                fib2 = par (parFib (n-2) t);
                fib1 = par (parFib (n-1) t)
              in (sync fib2) + (sync fib1);
```

This new version of `parFib` takes an extra argument that controls the amount of parallel tasks created: if the argument supplied to `parFib` is smaller than the threshold `t`, then `seqFib` is used instead of `parFib`.

2.4 Example 2: The `parMap` Skeleton

The `par` and `sync` primitives can be seen as low level constructs for parallel programming. Using these primitives it is possible to write more powerful abstractions for the *p*Fun language, such as Algorithmic Skeletons [6]. Algorithmic skeletons are higher-order functions that encapsulate common patterns of parallel computation.

For example, the function `map`, common in widely-used functional languages, is a higher-order function that takes two arguments, a function and a list, and applies the function to every element of the list generating a new list. A parallel `map` is a function that has the same type as the sequential `map` but applies the function argument to every element of the list in parallel.

To implement a parallel `map` in *p*Fun, we first need a function to create parallel tasks:

```
parList:: (a->b) -> [a] -> [Par b]
parList f l = case l of
        [] -> [];
        (x:xs) -> (par (f x)) : (parList f xs);
```

The function `parList` creates a list of possible parallel tasks, and we need a way of accessing the values computed by these tasks:

```
syncList :: [Par a] -> [a]
syncList list = map sync list;
```

Finally, the `parMap` skeleton is implemented using the functions `parList` and `syncList`:

```
parMap :: (a->b) -> [a] -> [b]
parMap f l = syncList (parList f l);
```

Parallelism arises because `parMap` first uses `parList` to create the `Par` objects, before `syncList` is used. If there are processors available they will start executing these `Par`s. Once all `Par`s are created, `syncList` is used to collect the results. When `syncList` tries to consume the list produced by `parList`, it will either block waiting for the result of a `Par` that is already being computed (in that case the processor executing `syncList` will get another task to execute), or will start executing one of the `Par` objects in the list.

3 The *p*Fun Compiler and Runtime System

3.1 Distributed Scheduling

The *p*Fun runtime runs a dynamic scheduler implementing a work stealing strategy. This scheduling is particularly interesting to exploit the inherent nested fork-join program structure obtained by the `par`/`sync` parallel constructors.

Work stealing is a general denomination of a receiver-initiated class of distributed load balancing schedulers. The basic algorithm assumes that the responsibility for managing and executing the set of tasks generated by a running application is shared among processors of a parallel machine. The number of tasks executing simultaneously on each processor is limited in order to allow each processor to maintain a reserve of work represented by a ready tasks queue. The scheduler uses the length of ready queues as the load information. Also distributed among the processors is the control for scheduling decisions. Depending on the size of such queues a processor can start a scheduling operation. A processor sends a request for new ready tasks (a work steal) to another randomly chosen processor when its local ready queue reaches a value below of a certain limit. When a processor receives a steal message it will answer with a task taken from its local queue if the amount of work in reserve is above a certain limit; otherwise the message is forwarded to another randomly chosen processor. *p*Fun's runtime system was mainly inspired by GUM (Graph reduction for a Unified Machine model) [25], a distributed virtual machine that implements GpH [24], a Haskell extension for parallel programming. Local thread scheduling and synchronization are implemented in a similar way to GUM. The main differences are on the way work is distributed between nodes and on the portability of the system. The *p*Fun runtime system was designed to be more adaptable to heterogeneous environments. Programs are compiled into byte-code, and the runtime system is implemented using standard C and TCP/IP sockets for communication, maintaining a high degree of portability. The current prototype implementation of *p*Fun uses a simpler distributed model than GUM, based on *work servers* and *slaves*. The slave nodes are dedicated to execute tasks while work servers can also answer steal requests. The `par` primitive adds a task to the ready queue of a work server. Slaves are hosts that connect to a work server asking for computations to execute. A slave receives work, executes it, and sends the result back to its server. Slaves keep no state: a message containing work carries all the code and data needed to execute it. That is interesting for large scale networks since the code for the application does not need to be pre-loaded on all hosts. Having the *p*Fun system installed on all locations, we can start a *p*Fun program on one host and it will spread to all locations. GUM is implemented on top of PVM and is designed to run on Beowulf clusters, hence all PEs (Processing Elements) know each other and they all function as work servers. GUM is also a closed system: all PEs must have the machine code for the application and once the system is running, no other PE can join the computation, while in *p*Fun's model, slaves and work-servers can join the computation at anytime. These differences also allow a higher degree of fault-tolerance in *p*Fun: In GUM, if one of the PEs dies the whole computation has to stop as the state of the computation is shared by all PEs. If a work server detects that a slave died it could send the task that was being evaluated by the dead slave to another one (although in the current system, if a work server dies, it can not be recovered). The fact messages contain the code and data needed to execute tasks allows slaves to be executed in

Bag-Of-Tasks GRID middlewares like OurGrid [5], something that we want to explore in the future.

*p*Fun's simpler distributed model and the compilation into byte-codes allow the system to be used on heterogeneous environments with the cost of a poor performance on divide-conquerer parallel applications (as can be seen in Section 4). As future work, we plan to modify the current prototype to better exploit the characteristics of different parallel architectures (Section 6).

3.2 The compiler

The current prototype implementation of *p*Fun supports a subset of Haskell's syntax with `let`s, algebraic data types and pattern matching (as in the examples given in section 2). Programs are compiled into a set of supercombinators (functions with no free variables) in the way described in [18]. These supercombinators are compiled into architecture independent byte-code (each supercombinator generates a byte-code file). Having no free variables makes it easier to implement the serialization algorithms as there are no environments to be communicated. As the subset of Haskell implemented is accepted by any Haskell compiler, type checking of parallel programs is done using GHCi [10] using the sequential definitions of `par` and `sync` given in section 2. We plan in the future to have our own implementation of type checking, specially to investigate the automatically insertion of the `sync` primitive described in section 2.

3.3 The Byte-Code Interpreter

The byte-code interpreter is an implementation of the SECD machine [15] using only one stack, and stack frames to separate nested function calls. The interpreter, at run time, allocates its internal structures in a garbage collected heap (section 3.6).

3.4 Distributed Execution

Even though *p*Fun is a *strict* functional language, in which the arguments of functions are evaluated before function application, the `par` primitive has a different semantics. The `par` primitive creates a *suspension* of its argument in the heap. A suspension represents an unevaluated expression. Suspensions are also used to implement partial applications of curried functions. For example, the following expression:

```
par (f arg1 arg2 ... argn)
```

will create in the heap the suspension in Figure 1. It contains a header that describes its type and layout, i.e., how many pointers to other heap objects it has. A suspension always contains a pointer to a function and an array of pointers to the arguments to be consumed by the function when the computation is started. Once created, the suspension is attached to a `Par` object and added to a ready queue. The `Par` object (Figure 2) is the value returned from a call to `par`.

A `Par` object can be in three states:

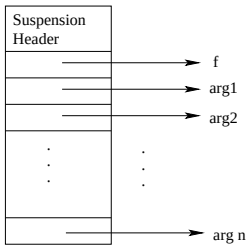


Fig. 1. A suspension in the Heap

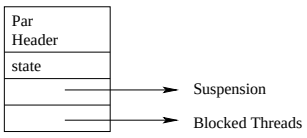


Fig. 2. A `Par` Object

- `Not Evaluated`: In this case it can be sent to a slave to be evaluated remotely or it can be evaluated locally when the scheduler of the byte-code interpreter runs out of runnable threads
- `Being Evaluated`: It is being evaluated either locally or by a remote processor
- `Evaluated`: It does not point to a suspension anymore, but to the result of the evaluation of a suspension

A call to `sync` always receives a `Par` object as its argument and it takes an action based on the state of `Par`. If the `Par` object is in the `Not Evaluated` state, `sync` will create a local thread to evaluate the `Suspension` pointed by the `Par`. The current thread will be blocked and added to a list of blocked threads on that `Par` object. Once the evaluation of the suspension is finished, the `Par` object is updated to point to the result of the computation, and all threads blocked on it will be added to the pool of runnable threads. When `sync` finds a `Par` that is `Being Evaluated`, it blocks the current thread and adds it to the list of blocked threads on that object. If `sync` finds a `Par` that is `Evaluated`, it simply returns the result of the computation pointed by it.

The Scheduler of the byte-code interpreter executes the following loop:

1. If there are runnable threads, execute one of them
2. If there are no runnable threads, create a new thread to evaluate a `Par` object from the ready queue
3. If the ready queue is empty, look for work on a work server

As in GUM, threads are never preempted and are executed until they complete or until they block waiting for a value being computed by another thread. This has the advantage of tending to decrease space usage and overall runtime [25].

When looking for work, the scheduler uses a simple protocol with the following messages:

- `NeedWork`: Message sent to a work server asking for work
- `Work`: This message contains a serialized task to be executed by a slave or work server. This message is sent by a work server to a slave or another work server as an answer to a `NeedWork` message. A `Work` message contains all the code and data needed to execute the task on a remote host

– `NoWork`: This message is the answer to a `NeedWork` message when the work server has no work to give. In that case, the Scheduler will wait for some time and reissue another `NeedWork` message later or send it to another work server
– `Result`: A result message contains the result of the evaluation of a task. It is the answer to a `Work` message. When a `Result` message is received, the original `Par` object must be updated with the result of the evaluation and all threads blocked on that `Par` are added to the runnable pool.

When a work server receives a `NeedWork` message, it checks if there are available tasks in its ready queue. If so, it sends the task (a suspension) back to the slave, using a `Work` message, where it will be evaluated.

To evaluate a suspension, a new thread is created: A thread object is allocated in the heap, and the arguments of the expression are pushed onto the thread's stack. The byte-code evaluator then jumps into the code for the function. Once the thread finishes evaluation, the result of the evaluation will be on top of the stack.

3.5 Serializing expressions

The graph representing the computation being communicated is packed at the source and unpacked at the destination. Packing, or *serializing*, arbitrary graph structures is not a trivial task and care must be taken to preserve sharing and cycles. Packing in *p*Fun is done exactly in the same way as in GUM [25].

3.6 Garbage Collection

The current prototype implementation of the *p*Fun runtime system has a *generational* garbage collector [12] for local GC. Garbage collection occurs locally at each site and the roots for garbage collection are the stacks for all threads and the ready queues. *p*Fun's garbage collector is parallel in the sense that garbage collection occurs locally and independently at each site.

The main difference between a traditional garbage collector and the one used in *p*Fun is in the way it deals with `Par` objects: if during the evaluation of a task by a slave GC occurs in the work server, when the result is back, the heap address of the original task is not valid anymore. Therefore, the `Par` object to be updated with the result of the computation can not be found.

To solve this problem, when a `Par` object is created in the heap, it is given a *stable address* and we keep a table that maps stable addresses to actual heap addresses. When garbage collection occurs, this table must be updated so that stable addresses always point to the new heap addresses of `Par`s.

When a thread is sent to be evaluated on a remote location, its stable address is sent together with it. When a client sends a `Result` message back to the server it also contains the stable address of the original `Par`. The work server then finds the original `Par` through the table of stable addresses, and updates it with the result of the remote evaluation.

4 Preliminary results

The first set of experiments in this section were performed on 8 computers, each with an AMD Athlon(tm) XP 2400+ processor and 192MB of RAM, using one as a work server and the other 7 as slaves. Table 1 shows the run times for 5 different programs. The speedups reported here and throughout this section are *relative*, i.e., improvement over the single-processor parallel execution. `parFib` is the program given in section 2.3. `pTak` [14] is a parallel version of the Takeuchi function. It is similar to `parFib`: parallelism is introduced at each recursive call and we use a threshold to control the amount of parallelism. The third program is `parMapFib35` calculates 8 times the `seqFib` of 35. As `parMap` is used, this program generates 8 threads that can be evaluated in parallel, one for each element of the list. The `pMaze` program searches for an exit in a maze. The maze is represented as a tree and we use depth first search to find the exit. Parallelism is introduced with `parMap`. `pCoins` is a more realistic program: given a collection of coins and an amount to be paid, it computes the number of possible ways to pay it. It uses a divide-and-conquer algorithm and parallelism is again introduced with the `parMap` skeleton.

Table 1. *p*Fun on 8 nodes (1 work server and 7 slaves)

	1 Proc (sec)	2 Proc (sec)	4 Proc (sec)	6 Proc (sec)	8 Proc (sec)
`parFib`	58.7	27.7	31.8	15.9	21.0
`pTak`	44.2	25.6	26.2	28.5	31.5
`parMapFib35`	103.1	51.3	31.2	26.8	15.6
`pMaze`	46.7	23.2	13.9	9.3	9.3
`pCoins`	48.5	35.9	36.1	36.2	37.5

Table 1 shows that, for the set-up used in the experiments, the distributed scheduling performed by *p*Fun's runtime system works better for data parallel programs (`parMapFib35` and `pMaze`) than for divide-and-conquer programs (`parFib`, `pTak` and `pCoins`). `parMapFib35` creates only 8 tasks, one for each computer, hence it improves run time for all number of processors measured. The same happens with `pMaze` that creates 10 threads of equal size, one to evaluate each branch of the tree. In `parFib` there is an improvement of performance up to 6 processors, after that there is an increase of communication in the system affecting performance: there are many idle slaves sending messages asking for work, and the work server's ready queue is empty. `pTak` generates lots of threads but most of the threads are created not in the work server but in the slaves, hence most of the time there are some heavy-loaded slaves while other slaves are idle and the work server's ready queue is empty. The performance of a divide and conquer algorithm could be improved if some of the slaves were substituted

by work servers connected to slaves. In that case the work generated by sub threads could be redistributed. Of course, there is a limit to that, and it depends on the structure of the application. The `pCoins` program creates 1 large grained thread, and many fine grained threads, therefore the runtime on more than one processor is always the time needed to evaluate the larger thread.

Table 2. Speedup on a Centrino Dual Core

	Speedup
`parfib`	1.75
`pmapFib35`	1.93
`pTak`	1.2
`pMaze`	1.96
`pCoins`	1.33

Another interesting result came from a different set-up: we used only one laptop computer with a Centrino Duo 1.60GHz processor, an Intel dual core processor for laptops, and only one work server and one slave, each allocated to a different core of the processor. For all parallel programs some speedup was achieved as can be seen in Table 2.

Table 3. Comparing Sequential Programs

	pFun (Fastest Parallel) (sec)	pFun (1Proc) (sec)	pFun (Seq) (sec)	GHCi (sec)	GHC (sec)	Caml (sec)
`seqFib`	15.9	58.7	53.9	177.9	18.6	14.6
`mapFib35`	15.6	103.1	105.0	335.3	34.1	27.3
`Tak`	25.6	44.2	45.0	112.8	12.7	17.4
`Maze`	9.3	46.7	46.5	151.5	16.0	11.9
`coins`	35.9	48.5	48.3	66.4	6.9	16.8

Table 3 compares *p*Fun's performance with two other compilers for functional languages. GHC [10] is an optimizing machine-code compiler for the Haskell *lazy* functional language. GHC also comes with an interpreter called GHCi that compiles Haskell programs into byte-code. *p*Fun is faster than GHCi for all programs measured. Haskell is a lazy functional language and its evaluation model is very different than the one of a strict language like *p*Fun (the different evaluation models for functional languages are explained in [8]). Caml-Light [3] is a fast optimizing byte-code compiler for the Standard ML *strict* functional language.

*p*Fun's sequential run times are 2.7 up to 3.9 times slower than Caml-Light. Caml-Light is a very fast implementation of Standard ML, and several optimizations are applied to the programs at compile time and runtime. *p*Fun is still a prototype, and almost no optimizations are applied to the generated byte-code. It is interesting to notice that *p*Fun's fastest parallel execution is always faster than GHC and Caml-Light for the data parallel programs, and it is also faster than GHC for the fibonacci program.

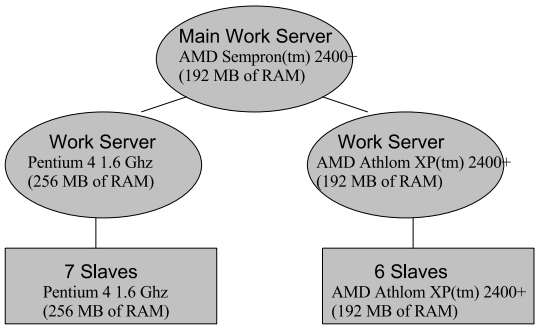


Fig. 3. The GRID used in the experiments

Finally, the next experiment was executed using a small GRID composed of three work servers connecting two Beowulf clusters as in Figure 3. The application used was a calculation of π using the Monte Carlo method. The application creates tasks in two levels using `parMap`: First it creates 30 parallel tasks, each calculating π using 300,000,000 points. Each of these tasks divides its work creating 15 tasks, again using `parMap`. The run time of the application on one machine of the cluster on the left of Figure 3 (a Pentium 4 1.6 Ghz) was 4 hours, 58 minutes and 13 seconds. Its run time on one machine of the cluster on the right (a Athlon XP 2400+) was 2 hours, 49 minutes and 20 seconds. Using the GRID of Figure 3 the run time was 27 minutes and 18 seconds which is a speed up of **11** compared to the slow machines and of **6.2** compared to fast ones.

The tasks created using the first `parMap` are all added to the ready queue of the Main work server, and the two other work servers compete for these tasks. Each of these tasks will generate tasks to be evaluated by its local slaves. One important thing to notice is that during most of the time all computers of both clusters were busy executing computations, which is the main objective of a GRID scheduler. Obviously, towards the end of the execution the ready queue of the main work server is empty and the fast cluster has no work, while the main work server waits for the slower cluster to finish a computation, therefore reducing speed up. To avoid this problem, a work server, towards the end of the execution, could replicate computations on clusters that are recognized to be fast if they are idle, instead of waiting for slow clusters. *p*Fun computations can be replicated at any time as they are free of side effects.

5 Related Work

The potential of functional programming languages to support parallelism has been recognized for a long time and several extensions for parallel programming in functional languages have been implemented (for a survey on the field, the reader should refer to [11]). Here we discuss the ones that are more closely related to *p*Fun.

The `par` and `sync` primitives provide the same abstraction as *futures* in Multilisp [20]. The idea of futures is old but it is nowadays being adopted in many modern languages, e.g., [9, 22, 7]. pFun's `par` only indicates potential parallelism. In Alice [22], when a future is created a new thread is started automatically to evaluate the computation. In Multilisp, after the call to `future X`, the new task created to evaluate X is maintained active while the parent task is moved to a pending queue [20]. The creation of a future in Multilisp is a very expensive operation [20, 21], while `par` is only a *mark* saying that a expression could be evaluated in parallel with the current expression either locally or on a remote computer. In *p*Fun the creation of a `Par` object is cheap: its like creating a suspension and adding a pointer to the ready queue. Creating a suspension is a cheap operation: as can be seen in Figure 1 a suspension is just like a cons cell.

GPH [24] is a parallel extension of Haskell for parallel programming. To express parallelism, the programmer uses a `par` combinator (similar to *p*Fun's `par`). Since Haskell is a lazy language, it is difficult to predict the order of evaluation of expressions, thus the `seq` combinator must be used to control sequencing. Furthermore, as Haskell is a lazy language, programmers have to force the evaluation of expressions using *evaluation strategies*.

GRID/ML [1] is an extension of Standard ML for GRID programming. It runs on top of the ConCert network [4], a peer-to-peer network of interconnected nodes, each running the same abstract machine. All nodes maintain a queue of pending work, and they can steal work from other nodes. GRID/ML provides primitives (similar to *p*Fun's primitives) to express parallelism and populate the node's queue of pending work. The GRID/ML system focuses mainly on fault-tolerant distributed programming: GRID applications are written as a series of deterministic functions that can be memorized by the network and restarted at any time. No measurements of their current implementation are given.

The work stealing scheduling strategy is widely used in practical applications, in particular we name the Cilk [2] programming environment. Cilk requires an extra effort by programmers: programs must be developed in a nested fork-join structure. Different from *p*Fun, this programming effort is unnatural since Cilk extends an imperative programming language (C).

6 Conclusions and Future Work

We have presented the design and implementation of *p*Fun, a strict parallel functional language. *p*Fun provides a programming model based on annotations. Task creation, synchronization and load distribution are automatically managed by

its runtime system. Preliminary measurements for the prototype implementation show that the work-stealing mechanism used for load balancing performs better for data parallel programs. As future work we want to adapt *p*Fun's runtime system to better exploit the characteristics of each parallel architecture. Work messages contain code and data to be executed on slaves. That imposes a high overhead when executing on multi-core machines as all code is already loaded in memory. For clusters, a work server could keep track of the code that was already sent to a slave. This would reduce significantly the size of messages in applications that use `parMap` for example. Although not yet implemented, the current implementation of the runtime system could provide a high degree of fault tolerance. In the case of failure of one node, purely functional expressions can be re-started at any time as they are free of side effects. Currently, all messages contain the total heap size needed to unpack work. If a slave does not have enough space it could reject the work message and ask for another one. Work servers could also ping slaves to see if they are still alive, and in the case of failure, computations could be restarted on a different slave. At the language level, we plan to investigate how *p*Fun's programming model could be extended with higher-level abstractions to express parallelism, e.g., using different parallel skeletons. The performance shown in Section 4 are promising since we have obtained some speedup for different classes of architectures. As slaves do not distribute work, the current load distribution mechanism performs poorly for divide and conquer applications: towards the end of execution we have heavily loaded slaves, while some slaves are idle and the ready queue of the work server is empty. We plan to investigate the use of a distributed shared ready queue, where all locations can share work. This distributed queue could be implemented using a distributed shared memory as in the MT System [16] or GUM [25].

Acknowledgements The authors would like to thank CNPq/Brazil for the financial support provided.

References

1. T. Murphy VII. ML Grid programming with Concert. In *The 2006 ACM SIGPLAN Workshop on ML (ML 2006)*. ACM press, 2006.
2. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. C. E. Zhou. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, Aug. 1995.
3. The Caml-Light System. WWW page, http://caml.inria.fr/, 2005.
4. B. Chang, K. Crary, M. Delap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning. Trustless GRID computing in ConCert . In *Grid 2002*. Springer-Verlag, 2002.
5. W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.
6. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

7. J. Duffy and E. Essey. Running queries on multi-core processors. WWW page, MSD Magazine, http://msdn.microsoft.com/en-us/magazine/cc163329.aspx, 2008.
8. A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.
9. Futures. WWW page, http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/-FutureTask.html, 2008.
10. The Glasgow Haskell Compiler. WWW page, http://www.haskell.org/ghc, 2005.
11. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, UK, 1999.
12. R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic dynamic memory management*. John Wiley and Sons, 1996.
13. S. P. Jones and S. Singh. A tutorial on parallel and concurrent programming in haskell. In *Advanced Functional Programming Summer School*, To appear in LNCS. Springer-Verlag, 2008.
14. S. B. Junaidu. *A Parallel Functional Language Compiler for Message-Passing Multicomputers*. PhD thesis, University of St Andrews, 2003.
15. P. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
16. M. T. Morazán, D. R. Troeger, and M. Nash. *Paging in a Distributed Virtual Memory*, volume 3 of *Trends in Functional Programming*, pages 75–86. Intellect Books, Bristol, UK, 2002.
17. S. Peyton Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 1(13), 2003.
18. S. P. Peyton Jones and D. Lester. *Implementation of Functional Programming Languages*. Prentice Hall, 1987.
19. An Operational Semantics for pFun, a semi-explicit parallel functional language. WWW page, http://atlas.ucpel.tche.br/ dubois/pfunsem.pdf, 2008.
20. J. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
21. J. Robert H. Halstead. An assessment of Multilisp: lessons from experience. *Int. J. Parallel Program.*, 15(6):459–501, 1986.
22. A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, Feb. 2006.
23. H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.
24. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.
25. P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a portable implementation of Haskell. In *PLDI*, Philadephia, May 1996.

Realizing Multiparadigm Programming based on Hierarchical Graph Rewriting

Petra Hofstedt[1] and Kazunori Ueda[2]

[1] Dept. of Electrical Engineering and Computer Science
Technical University of Berlin
ph@cs.tu-berlin.de
[2] Dept. of Computer Science, Waseda University
ueda@ueda.info.waseda.ac.jp

Abstract. The paper presents the new multiparadigm programming language CCFL which allows the description of concurrent processes and of non-deterministic behavior and it discusses the compilation of CCFL programs into the hierarchical graph rewriting language LMNtal.
LMNtal aims at the unification of paradigms of computation and supports by its features and structure the transformation of CCFL programs such that we reached clear and simple compilation schemata.

1 Introduction

The concurrent, hierarchical graph rewriting language LMNtal [16, 17] aims to unify various paradigms of computation resp. computational models. It has been used for encoding various calculi [15, 14], e.g. the pure lambda calculus and the ambient calculus. In this paper, we show the use of LMNtal as base model and target language for the compilation of programs of a new multiparadigm programming language CCFL.

The *C*oncurrent *C*onstraint-based *F*unctional *L*anguage CCFL combines concepts from the functional and the constraint paradigms. CCFL allows besides the description of deterministic computations using a functional programming style also the implementation of non-deterministic behavior based on constraints. Moreover, CCFL enables to describe systems of concurrent processes, whose communication and synchronization is based on the concurrent constraint programming (CCP) model [12].

The compilation of CCFL programs into LMNtal rules is based on adaptions of translation techniques for functional into logic languages. CCFL data elements and variables are encoded by means of heap data structures handled during run-time in LMNtal. The introduction of these data structures allows us to deal with free variables and constraints in CCFL, to realize lazy evaluation, and it even simplifies the LMNtal rule-set generated from the compilation for handling higher-order functions and partial applications.

Encoding CCFL based on LMNtal, we were able to underline three things: 1) LMNtal is well suited as base for the encoding of calculi *and* programming

languages, 2) it supports particularly the combination of concepts of different paradigms and, thus, the realization of a multiparadigm language, and 3) this is, accordingly, reachable by relatively simple compilation schemata.

Related Work LMNtal has been used for the encoding of various calculi, as presented by Ueda in [15, 14] e.g. for the pure lambda calculus and the ambient calculus, resp. Like in the approach presented in this paper, the membrane construct of LMNtal plays an essential role for the clarity and simplicity of the encoding.

There are several functional languages allowing for concurrent computation of processes among them EDEN [8] and ERLANG [1], both using explicit notions for the generation of processes and their communication and CONCURRENT HASKELL [11] which supports threads via the IO monad.

The language GOFFIN [2] combines HASKELL with a constraint-based coordination language to express parallelism and non-deterministic computation. It provides a similar structure like CCFL, while CCFL's constraint abstractions are more oriented to predicates than to functions and the ask-constraint's functionality is a bit more extended. Moreover, in [5] we discuss the extension of CCFL constraints to typical constraint systems.

CURRY [4] is a functional-logic language combining the functional and the logic paradigms and builds on the evaluation principle narrowing in contrast to residuation and non-deterministic choice in CCFL.

Beside the realization of CCFL based on hierarchical graph rewriting as discussed in this paper, there exists a CCFL implementation for a parallel multicore architecture which supports the realization of data and task parallel skeletons as presented in [6].

2 The Language CCFL

The multiparadigm programming language CCFL combines concepts and constructs from the functional and the constraint-based paradigms. A CCFL program consists of data-type definitions, functions and user-defined constraints. Functions are used to express deterministic computations, while user-defined constraints allow the description of cooperating processes and non-deterministic behavior.

2.1 Functional Programming

CCFL's functional sublanguage is a lazy language with polymorphic type system. A function consists of a type declaration and a definition allowing the typical constructs such as case-expressions, let-expressions, function application, and some predefined infix operator applications, constants, variables, and constructor terms. A function call (without free variables) evokes a computation by reduction. Prog. 2.1 shows the definition of a polymorphic data-type *List a* and a function *length* computing the length of lists. In the following, we will use

Program 2.1 List length

```
data List a = [] | a : (List a)

fun length :: List a -> Int
def length list = case list of []   -> 0;
                               x:xs -> 1 + length xs
```

the HASKELL-typical notions for lists, i.e. [] and e.g. [1,2,4] for an empty and non-empty list, resp., and ":" as the list constructor.

Free Variables Expressions in CCFL may contain free variables. Function applications with free variables are evaluated using the residuation principle [13], that is, function calls are suspended until the variables are bound to expressions such that a deterministic reduction is possible. For example, a function call $(4 + x)$ with free variable x will suspend. On the other hand, the application $(length\ [y,1,x])$ successfully evaluates to 3 because the values of the free variables y and x are not necessary to proceed with the computation.

2.2 Constraint-based Programming with CCFL

A user-defined constraint is given by its declaration and a constraint abstraction. A constraint abstraction consists of a head and a body which may contain the same elements as a function definition. Additionally, the body can be defined by several alternatives the choice of which is decided by guards. A constraint abstraction is allowed to introduce free variables and each body alternative is a conjunction of constraint atoms. A constraint always has result type ***C***.

ask- ***and*** **tell-*constraints*** Within a CCFL rule constraints may have two functionalities: *tell*-constraints generate concurrently working processes which may create knowledge in form of variable bindings (or constraints in general[3]). These processes may communicate over common variables. In contrast, *ask*-constraints do not generate knowledge but check for concrete variable bindings or constraints. *ask*-constraints control the choice of (potentially competing) rules and, thus, allow to express the synchronization of concurrently working processes on the one hand and non-deterministic computations on the other hand.

User-defined constraints mainly serve two purposes: expressing concurrent computations and dealing with non-determinism.

Concurrent Processes CCFL allows the description of systems of communicating and cooperating processes. Consider as an example Prog. 2.2 defining a producer and a consumer communicating over a common buffer.

[3] In [5] we discuss the extension of CCFL by constraints of other domains, e.g. arithmetic or finite-domain constraints.

Program 2.2 A producer-consumer setting

```
1  fun produce :: List a -> C
2  def produce buf =
3        with buf1 :: List a , item :: a
4        in ...       -- generate item here then
5                     -- put it into the buffer and continue
6           buf =:= item:buf1 & produce buf1
7
8  fun consume :: List a -> C
9  def consume buf =
10       buf =:= first:buf1 ->
11          ...               --  consume first here
12          consume buf1      --  and continue
13
14 fun main :: C
15 def main = with buf :: List a
16             in produce buf & consume buf
```

The *main*-function creates the buffer as a fresh variable *buf* using the with-construct and initiates the computation. The constraint applications *produce buf* and *consume buf* create concurrently working producer and consumer processes. The user-defined constraint *produce* (lines 1–6) describes the behavior of the producer process. It generates buffer elements *item*, puts them into the buffer using the equality constraint *buf* =:= *item* : *buf1* and concurrently initiates the computation of the remaining buffer *buf1* (lines 4–6). The consumer process must wait until the buffer has been filled with at least one element *first* . This is ensured by the match-constraint *buf* =:= *first* : *buf1* (line 10) of the guard of the *consume* rule. Note that, in contrast, the producer process is not restricted to synchronize with the consumer because the *produce*-rule is not guarded.

The constraints in the body of the rules are *tell-constraints.* They create processes which may compute bindings for the incorporated variables. Several *tell*-constraints combined by the &-combinator (as in line 6) generate an according number of processes and they communicate over common variables. *tell*-constraints are either applications of user-defined or predefined constraints, e.g. *produce buf1* in line 6, or they are equality constraints $x =:= fexpr$ between a variable x and a functional expression *fexpr* (also see line 6). Equality constraints are interpreted as strict. That is, the constraint $t_1 =:= t_2$ is satisfied, if both expressions can be reduced to the same ground data term [4]. While a satisfiable equality constraint $x =:= fexpr$ produces a binding of the variable x to the functional expression *fexpr* and terminates with result value *Success*, an unsatisfiable equality is reduced to the value *Fail* representing an unsuccessful computation.

The atoms of the guard of a user-defined constraint are *ask-constraints.* If a guard of a rule with matching left-hand side is entailed by the current accumulated bindings (and constraints in general), the concerning rule alternative

Program 2.3 A simple game of dice

```
1  fun game :: Int -> Int -> Int -> C
2  def game x y n =
3    case n of 0 -> x =:= 0 & y =:= 0 ;
4              m -> with x1, y1, x1, y2 :: Int
5                   in dice x1 & dice y1 &
6                      x =:= x1 + x2 & y =:= y1 + y2 &
7                      game x2 y2 (m-1)
8
9  fun dice :: Int -> C
10 def dice x =
11   member [1,2,3,4,5,6] x
12
13 fun member :: List a -> a -> C
14 def member l x =
15     l =:= y:ys -> x =:= y |
16     l =:= y:ys -> case ys of []   -> x =:= y ;
17                              z:zs -> member ys x
```

may be chosen for further derivation. In case that the guard fails or cannot be decided (yet), this rule alternative is suspended. If all rule alternatives suspend, the computation waits (possibly infinitely) for a sufficient instantiation of the concerning variables.

For *ask*-constraints, we distinguish between bound-constraints ***bound*** x checking, whether a variable x is bound to a non-variable term, and match-constraints (e.g. line 10) $x =:= c\ x_1 \ldots x_n$ which test for a matching of the root symbol of a term bound to the variable x with a certain constructor c. The variables $x_1 \ldots x_n$ are fresh.

Non-deterministic Computations *Ask*-constraints support the programming of non-deterministic behavior. For an example consider Prog. 2.3. The constraint *game x y n* initiates a game between two players throwing the dice n times and reaching the overall values x and y, resp.

The *tell*-constraints *dice x1* and *dice y1* (line 5) non-deterministically produce values which are consumed by the equality constraints $x =:= x1 + x2$ and $y =:= y1 + y2$, resp. Note that their computation is suspended until the arguments are (sufficiently) instantiated to apply the built-in function +.

The constraint abstraction *member* is the source of non-determinism in this program. It chooses a value from a list. Since the match-constraints of the guards of both alternatives are the same (lines 15 and 16), i.e. $l =:= y : ys$, the alternatives are chosen non-deterministically which simulates the dice.

Program 3.1 LMNtal: encapsulating computations by membranes

```
1  {@r,{$p},$s} :- {{@r,$p},$s}.
2  {{@r,$q}/,$s} :- {@r,$q,$s}.
3
4  {append([],Y,Z) :- Y = Z.
5   append([XH|XR],Y,Z) :- Z = [XH|ZR], append(XR,Y,ZR).
6   {append([1,2],[4],R)}, {append([9],[8,1],P)}
7  }
```

3 LMNtal

We briefly introduce the language LMNtal which is the target language of the compilation of CCFL programs. LMNtal is a concurrent language based on hierarchical graph rewriting. One of its major goals is to unify various paradigms of computation resp. computational models [16]. Thus, we chose LMNtal as target language for the compilation of the multiparadigm programming language CCFL.

An LMNtal program describes a process consisting of atoms, cells, links, and rules. Links connect atoms and/or cells to build graphs. Cells are processes encapsulated by membranes and allow to represent hierarchies in graphs. Rules are used to describe graph rewriting.

Consider Prog. 3.1. Lines 4–7 show a cell which is encapsulated by a membrane "{}". It contains two *append*-rewrite rules in a PROLOG-like syntax and two cells each enclosing an *append*-atom in line 6. *Y*, *Z*, *XH*, ... are links. In the current situation, the *append*-rewrite rules cannot be applied on the *append*-atoms in line 6 because they are enclosed by extra membranes.

The rules in lines 1 and 2 realize the unpacking of the *append*-atoms from their membranes for reduction. At this, *@r* denotes a (multi)set of rules, the so-called rule-context, and *$p* and *$s* are process-contexts, i.e. they stand for (multi)sets of cells and atoms. The template {*@r*, *$q*}/ in the second line has a stable flag "/" which denotes that it can only match with a stable cell, i.e. a cell containing no applicable rules.

In the current situation, the rule in line 1 is applicable to the cell of the lines 4–7, where *@r* matches the *append*-rules, *$p* matches one of the inner *append*-atoms and *$s* stands for the remaining cell. A possible reduction of the this cell is, thus, the following:

```
{append([],Y,Z) :- Y = Z.
 append([XH|XR],Y,Z) :- Z = [XH|ZR], append(XR,Y,ZR).
 {append([1,2],[4],R)}, {append([9],[8,1],P)}}
⇝(1)
{{append([],Y,Z) :- Y = Z.
  append([XH|XR],Y,Z) :- Z = [XH|ZR], append(XR,Y,ZR).
  append([9],[8,1],P)},
 {append([1,2],[4],R)}}
```

$\rightsquigarrow^*_{append}$
$\{\{\, append\,([\,]\,, Y, Z)\ :-\ Y = Z.$
$\quad append\,([\,XH|\,XR]\,, Y, Z)\ :-\ Z = [\,XH|\,ZR]\,,\ \ append(XR, Y, ZR)\,.$
$\quad P = [\,9\,,8\,,1\,]\,\}\,,$
$\ \{\, append\,([\,1\,,2\,]\,,[\,4\,]\,,R)\}\}$

The upper inner cell is now stable such that no rule is applicable inside. Thus, we can apply the second outer rule (line 2).

$\ldots$
$\rightsquigarrow_{(2)}$
$\{\, append\,([\,]\,, Y, Z)\ :-\ Y = Z.$
$\ append\,([\,XH|\,XR]\,, Y, Z)\ :-\ Z = [\,XH|\,ZR]\,,\ \ append(XR, Y, ZR)\,.$
$\ P = [\,9\,,8\,,1\,]\,,$
$\ \{\, append\,([\,1\,,2\,]\,,[\,4\,]\,,R)\}\}$

In this state, again the first outer rule (line 1) is applicable which yields after a number of steps the following stable state:

$\{\, append\,([\,]\,, Y, Z)\ :-\ Y = Z.$
$\ append\,([\,XH|\,XR]\,, Y, Z)\ :-\ Z = [\,XH|\,ZR]\,,\ \ append(XR, Y, ZR)\,.$
$\ P = [\,9\,,8\,,1\,]\,,\ \ R = [\,1\,,2\,,4\,]\,\}$

As one can see by the above example, LMNtal supports a PROLOG-like syntax. However, there are fundamental differences.

The above example already demonstrated the use of process-contexts, rule-contexts, membrane enclosed cells, and the stable flag. Different from other languages, the head of a rule may contain several atoms, even cells, rules, and contexts. A further important difference to other declarative languages are the logical links of LMNtal. What one may hold for variables in our program, e.g. *Y*, *Z*, *XH*, ... are actually links. Their intended meaning strongly differs from that of variables. Declarative variables stand for particular expressions or values and, once bound, they stay bound throughout the computation and are indistinguishable from their value. Links in LMNtal also connect to a structure or value. However, link connections may change. While this is similar to imperative variables, links are used to interconnect exactly two atoms, two cells, or an atom and a cell to build graphs and they have, thus, (at most) two occurrences. In rules, logical links must occur exactly twice.

Semantically, LMNtal is a concurrent language realizing graph rewriting. It inherits properties from concurrent logic languages. The rule choice is non-deterministic, but can be controlled by guards (not shown in the above example). As demonstrated above, the encapsulation of processes by membranes allows to express local computations and, it is possible to describe the migration of processes and rules between local computation spaces.

For a detailed description of LMNtal we refer to [7, 16, 17]. In Sect. 4, we discuss further examples of LMNtal programs as results of the compilation process of CCFL programs.

4 Encoding CCFL into LMNtal

CCFL integrates functional and constraint programming. Since constraints can be considered as particular functions, the code generation treats them uniformly such that we reach a proper unification of both paradigms. The transformation of CCFL functions and constraints is partially based on translation techniques [3, 9, 10, 18] for functional into logic languages.

4.1 Functional Elements

Let us first consider the functional sublanguage of CCFL. A CCFL function definition is translated into a set of LMNtal rules. At this, there is one initial rule and possibly a set of subordered rules realizing e.g. pattern matching as necessary for case-constructs.

CCFL data elements and variables are represented and processed by means of heap data structures during run-time. However, to clarify the presentation in this section, we represent CCFL variables directly by LMNtal links[4] instead and data structures by LMNtal atoms. We concentrate on the heap structures in Sect. 4.3 subsequently.

The right-hand side of a CCFL function definition is an expression composed by the typical constructs such as function applications, infix operator applications, let- and case-constructs and variables, constants, and constructors. CCFL infix operations are mapped onto their LMNtal counterparts. Function applications are realized by an atom *app* (...) and an according *app*-rule which is also used for higher-order functions as discussed in Sect. 4.2. Case-expressions generate extra LMNtal rules for pattern matching, let-constructs are straightforward realized by equality constraints.

Example 1. Consider Prog. 4.1 as compilation result of the list length function from Prog. 2.1.

The LMNtal program illustrates the generation of different rule alternatives from the case-construct and the handling of function applications and predefined infix operations. Also note the additional link argument *V0* of the *length*-rewrite rule. This link is used to access the result of the rule application which is necessary because LMNtal explicitly deals with graphs while a computation with a functional language like CCFL yields an expression as result.

4.2 Higher-order Functions and Partial Application

We use a transformation scheme from [18] to allow higher-order function application. For every CCFL function $f\, x_1 \ldots x_n = expr$ a rewrite rule $app\ (f,\ x_1, \ldots, x_n) :- f\ (x_1, \ldots, x_n)$ is generated which in combination with

[4] Moreover, we tolerate n-fold occurrences of links in rules, where $n \neq 2$. This is also not conform with LMNtal, where links must occur exactly twice in a rule, but the problem disappears with the introduction of heap data structures as well.

Program 4.1 LMNtal program as compilation result: list length

```
1  length (List, V0) :-
2    case__length (List, List, V0) .
3
4  case__length (V1, List, V0), nil (V1) :-
5    V0 = 0 .
6
7  case__length (V1, List, V0), cons (X, XS, V1) :-
8    V0 = 1 + V3, app (length, XS, V3) .
9
10 app (length, X1, X2) :-
11   length (X1, X2) .
```

the generation of an *app*-atom for a function application realizes higher-order function application.

The partial application of functions in CCFL is enabled by a number of additional LMNtal rules per CCFL function, where we adopt a transformation given in [3, 9]. To simplify the generation of rules for the partial applications of functions, the CCFL compiler performs an η-enrichment of functions, i.e. additional arguments are appended to the left-hand sides and to the right-hand sides of function definitions according to the function type declaration.

Example 2. Let *add* and *addOne* be CCFL functions, where the latter is defined by a partial application of the former.

```
fun add :: Int -> Int -> Int
def add a b = a + b
fun addOne :: Int -> Int
def addOne = add 1
```

The η-enrichment of the function *addOne* within the compilation process yields the following intermediate representation.

```
def addOne x = (add 1) x
```

The method to enable the partial application of functions from [3, 9] generates $\frac{n\times(n+1)}{2}$ rules for every n-ary function. In [5] we give a detailed presentation of the rule-set; we show an illustrative example here instead.

Example 3. Consider again the functions *add* and *addOne* from Example 2. A CCFL derivation sequence is the following:

$addOne\ 2 \rightarrow (add\ 1)\ 2 \rightarrow 1 + 2 \rightarrow 3$

The functions *add* and *addOne* are translated into LMNtal rules as discussed in Sect. 4.1. Additionally, we generate rules for each function for all possible cases of its partial application. These rules just generate constructor terms of

the function name and allow in this way to keep the data and to suspend the computation until the function can be fully applied.

```
add (A,B,V0) :- V0 = A + B .
addOne (V3,V4) :- app (add,1,X), app (X,V3,V4) .

app(add,V0) :- add(V0).                    // handling V0 = add,
app(add,V0,V1) :- add(V0,V1).              //   V1 = add V0,
app(add (V0),V1,V2) :- add(V0,V1,V2).      //   V2 = (add V0) V1,
app(add,V0,V1,V2) :- add(V0,V1,V2).        //   add as HOF arg.,
app(addOne,V0) :- addOne(V0).              //   V0 = addOne,
app(addOne,V0,V1) :- addOne(V0,V1).        //   addOne as HOF arg.
```

Using these LMNtal rules we rewrite the atom *addOne*(2,R) which corresponds to the CCFL expression *addOne* 2:

addOne(2,R)
$\rightsquigarrow$ *app*(*add*,1,X), *app*(X,2,R)
$\rightsquigarrow$ *add*(1,X), *app*(X,2,R) $\equiv$ *app*(*add*(1),2,R)
$\rightsquigarrow$ *add*(1,2,R)
$\rightsquigarrow$ $R = 1+2$
$\rightsquigarrow$ $R = 3$

The representation of data and variables by heap structures as run-time environment as discussed in Sect. 4.3 allows to fuse the set of $\frac{n\times(n+1)}{2}+1$ rules for handling partial applications and higher-order functions from the original approaches [3, 9, 18] into *one* unified rule.

4.3 Representing CCFL Expressions by Heap Structures

While links in LMNtal look similar to variables, they have a different functionality (cf. Sect. 3). In [5] we elaborately discuss that a direct translation of CCFL variables into LMNtal links (and, thus, of CCFL data structures into LMNtal atoms) would not be successful. Since links connect exactly two elements, i.e. cells or atoms, a representation of variables by links would disallow the representation of free variables and the sharing of data structures as needed for lazy evaluation. At least it can be shown [5] that a direct simulation is possible for a functional language (without constraints) using a call-by-value evaluation strategy, where multiply used data structures are just completely copied. Copying of data structures in LMNtal is supported, even if not very efficiently treatable in general.

The introduction of a heap as run-time environment enables the representation of free variables and the sharing of common data by different structures. This allows the compilation of user-defined constraints and rule guards and the implementation of evaluation strategies using sharing of data structures.

The generation and transformation of heap structures is directly incorporated into the generated LMNtal rules as result of the compilation process.

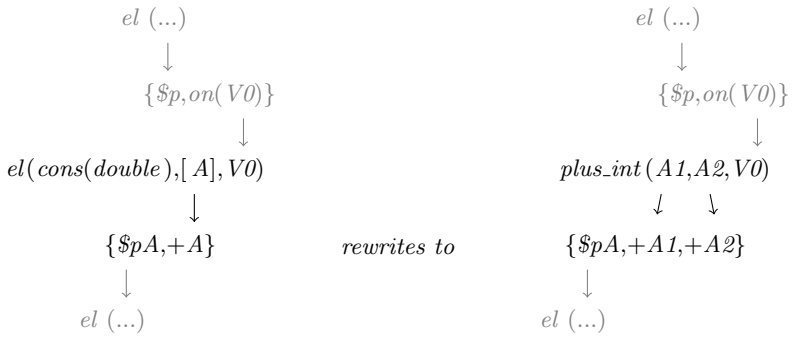


Fig. 1. A graph rewriting rule *double*

Example 4. Consider the arithmetic function *double*.

```
fun double :: Int -> Int
def double a = a + a
```

The compilation according to the simplified schema from Sect. 4.1 yields the following code.

```
double (A, V0) :- V0 = A + A .
```

Taking the handling of heap data structures into consideration we obtain the following rule, where the atom *plus_int* (*A1*,*A2*,*V0*) realizes a call to an application of the LMNtal infix operator +.

```
el (cons (double),[A], V0), {$pA,+A} :-
  plus_int (A1, A2, V0), {$pA,+A1,+A2} .
```

Fig. 1 visualizes the generated LMNtal rewrite rule. Non-variable CCFL-expressions[5] are represented by atoms $el(cons(F),OL,I)$, where the F is the function or constructor name, OL is a list of ($o$utgoing) links to cells connecting to the structures of the arguments according to the original CCFL function, and I is a ($i$ncoming) link from a cell which manages links onto the term. Accordingly, the CCFL term (*double a*) or $double(A,V0)$ in the intermediate LMNtal form, resp., yields the atom $el(cons(double),[A],V0)$. Variables are handled similarly. Cells containing links, like $\{\$p,on(V0)\}$, are used to connect between the atoms to build structures. In this way it is possible to realize sharing as illustrated by the cell $\{\$pA,+A1,+A2\}$ which hold two incoming links $+A1$ and $+A2$ and one outgoing link onto the common shared structure (indicated by the gray outgoing link onto the atom *el* (...)).

[5] This holds except for built-in functions, i.e. + is represented by a prelude rule *plus_int*.

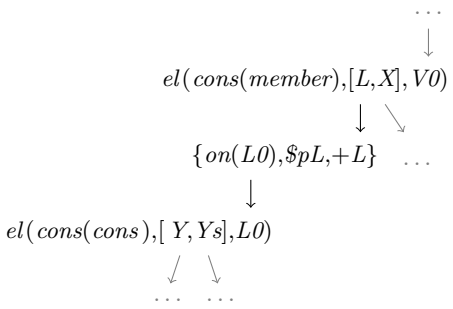


Fig. 2. A heap structure representing the expression *member* $(y : ys)\ x$

4.4 Constraints and Rule Guards

User-defined constraints and functions are handled uniformly in the compilation process. Constraints in LMNtal allow all the constructs as introduced for functions, but additionally free variables, a with-construct for the introduction of new variables within a rule, *ask*-constraints in the rule guard and *tell*-constraints in its body.

Just like LMNtal rules for CCFL functions, rules for user-defined constraints must hold an additional link in the rule head to connect to the computation result because constraints must be accessible as elements e.g. for partial application.

Ask-constraints, i.e. bound- and match-constraints are realized by additional atoms in the rule heads of the generated LMNtal code matching for the according heap structures.

Example 5. The user-defined constraint *member* of Prog. 2.3 non-deterministically chooses values from a given list. We either take the first element as result value or initiate a further computation on the rest of the list.

The guards consist of identical match-constraints $l =:= y : ys$ (lines 15 and 16) to realize a non-deterministic choice. The compilation yields, thus, identical rule heads matching the list structure $(y : ys)$ or $cons(Y, Ys)$, resp., as given below for both alternatives. The according heap structure is shown for illustration in Fig. 2.

```
el (cons (member) ,[L,X] , V0) ,
el (cons (cons) ,[Y, Ys] , L0) ,
{on (L0) , $pL,+L} :-
    ...
```

For *tell*-constraints we distinguish between applications of user-defined constraints which are just handled like function applications and equality constraints $t_1 =:= t_2$ (cf. Sect. 2.2). These latter base on a unification of the concerning subexpressions t_1 and t_2 of the constraint. A unification mechanism for heap data was implemented in LMNtal: *unify*(*L1*,*L2*,*R*) unifies the heap structures

Program 4.2 LMNtal compilation result: produce

```
1  el (cons (produce),[Buf],V0) :-
2    // generate a fresh variable buf1
3    el (var (...),OnBuf1,InBuf1), ..., {on (InBuf1),+Buf1},
4    // generate structure for item
5    ...
6    // generate expression (item:buf1)
7    el (cons (cons),[Item,Buf1],V3), {on (V3),+V2},
8    // unify call: buf =:= item:buf1
9    unify (Buf,V2,V0),
10   ...
```

connected to the links *L1* and *L2* and yields a result structure linked to *R*. The transformation of a CCFL equality constraint into concerning LMNtal code, thus, produces a *unify*-atom on both heap structures which initiates the unification process.

Example 6. Consider the producer-consumer example in Prog. 2.2 and the LMNtal rule for *produce* as compilation result given in Prog. 4.2.

The constraint $buf =:= item : buf1$ of the *produce*-constraint in line 6, Prog. 2.2, generates a concerning *unify*-atom in line 9, Prog. 4.2. This atom *unify* (*Buf*,*V2*,*V0*) initiates the unification of the two structures connected to the links *Buf* and *V2* which yields a result structure with link *V0*. While the first structure (connected to *Buf*) is passed as argument of the rule, the second structure (connected to *V2*) is generated by the code of lines 2-7 of Prog. 4.2.

4.5 Evaluation Strategies

LMNtal evaluates non-deterministically, and it does a priori not support certain evaluation strategies. Thus, the control of the order of the subexpression evaluation for CCFL is integrated into the generated LMNtal code. We realized code generation schemata for different evaluation strategies for CCFL by encapsulating computations by membranes using similar ideas as in Prog. 3.1.

We discussed the realization of evaluation strategies in more detail in [5]. Expressions are destructured into subexpressions which remain interconnected by links. Each subexpression is encapsulated by a membrane and builds a cell. Expressions to be reduced are provided with the rule-set such that their reduction becomes possible. The implementation of a call-by-value strategy requires the copying and provision of the generated LMNtal rule-set for every innermost redex to ensure that they are prioritized on the same level and independent reductions do not influence each other. For call-by-name and lazy evaluations we typically have one outermost redex and, thus, copying of the rule-set (which may become expensive) is not necessary.

5 Conclusion

We presented the multiparadigm programming language CCFL and its compilation into the hierarchical graph rewriting language LMNtal. We think that CCFL is a successful integration of the functional and constraint-based paradigms allowing a comfortable modeling of systems of concurrent processes and of deterministic and non-deterministic behavior.

The ability of the compilation target language LMNtal to unify and to model computation paradigms proved to be very useful for our compiler implementation. It could be shown that modeling combined language paradigms in LMNtal is possible in a convenient way and by means of clear transformations.

Acknowledgment The work of Petra Hofstedt has been supported by a postdoctoral fellowship No. PE 07542 from the Japan Society for the Promotion of Science (JSPS).

References

1. Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 2007.
2. Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, and Hendrik C. R. Lock. GOFFIN: Higher-Order Functions Meet Concurrent Constraints. *Science of Computer Programming*, 30(1-2):157–199, 1998.
3. Mantis H. M. Cheng, Maarten H. van Emden, and B. E. Richards. On Warren's Method for Functional Programming in Logic. In Peter Szeredi David H. D. Warren, editor, *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, ICLP 1990*, pages 546–560. MIT Press, 1990.
4. Michael Hanus, Sergio Antoy, Bernd Braßel, Herbert Kuchen, Francisco J. Lopez-Fraguas, Wolfgang Lux, Juan Jose Moreno Navarro, and Frank Steiner. Curry: An Integrated Functional Logic Language. Technical report, 2006. Version 0.8.2 of March 28, 2006.
5. Petra Hofstedt. CCFL – A Concurrent Constraint Functional Language. Technical Report 2008-08, Technische Universität Berlin, 2008. `http://iv.tu-berlin.de/TechnBerichte/2008/2008-08.pdf`, last visited 11 March 2009.
6. Petra Hofstedt and Florian Lorenzen. Constraint Functional Multicore Programming. In *4. Arbeitstagung Programmiersprachen – ATPS'09*, Lecture Notes in Informatics (LNI), 2009. to appear.
7. LMNtal PukiWiki. `http://www.ueda.info.waseda.ac.jp/lmntal/`. last visited 11 March 2009.
8. Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
9. Lee Naish. Adding equations to NU-Prolog. In *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP'91*, volume 528 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 1991.
10. Lee Naish. Higher-order logic programming in Prolog. Technical Report 96/2, Department of Computer Science, University of Melbourne, 1996.
11. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages – POPL*, pages 295–308, 1996.

12. Vijay A. Saraswat and Martin C. Rinard. Concurrent Constraint Programming. In *17th ACM Symposium on Principles of Programming Languages – POPL*, pages 232–245, 1990.
13. Gert Smolka. Residuation and Guarded Rules for Constraint Logic Programming. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming. Selected Research*, pages 405–419. The MIT Press, 1993.
14. Kazunori Ueda. Encoding Distributed Process Calculi into LMNtal. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 209:187–200, 2008.
15. Kazunori Ueda. Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting. In Andrei Voronkov, editor, *Rewriting Techniques and Applications – RTA*, volume 5117 of *Lecture Notes in Computer Science*, pages 392–408. Springer, 2008.
16. Kazunori Ueda and Norio Kato. LMNtal: a Language Model with Links and Membranes. In *Proceedings of the Fifth International Workshop on Membrane Computing (WMC 2004)*, volume 3365 of *LNCS*, pages 110–125. Springer, 2005.
17. Kazunori Ueda, Norio Kato, Koji Hara, and Ken Mizuno. LMNtal as a Unifying Declarative Language. In Tom Schrijvers and Thom Frühwirth, editors, *Proceedings of the Third Workshop on Constraint Handling Rules*, Technical Report CW 452, pages 1–15. Katholieke Universiteit Leuven, 2006.
18. D.H.D. Warren. Higher-order extensions to PROLOG: Are they needed? *Machine Intelligence*, 10:441–454, 1982.

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 our previous work on an expressive and elegant class of rewrite systems (called CERSs) that contains built-in numbers and supports the use of collection data structures such as sets or multisets, we consider context-sensitive rewriting with CERSs 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, we develop a dependency pair framework for context-sensitive rewriting with CERSs, 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.

1 Introduction

While ordinary term rewrite systems (TRSs) can be used for modeling algorithms in a functional programming style, there still remain 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. We have shown in [12] that constrained equational rewrite systems (CERSs) provide an expressive and convenient tool for modeling algorithms that solves both of these drawbacks. Since [12] considers only natural numbers as a primitive, the first contribution of this paper is a reformulation of

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

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.

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 [23, 25] 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 [24]). Furthermore, context-sensitive rewriting allows to model (certain aspects of) lazy evaluation as used in functional programming languages such as Haskell (for more on the relationship between lazy evaluation and context-sensitive rewriting, see [26]). In context-sensitive rewriting, the arguments where an evaluation may take place are specified 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 on the reduction strategy.

Example 1. Consider the following rewrite rules, where ins is used to add a further element to a set:

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

Here, the function symbol from is used to generate the (infinite) subsets of integers that are greater than or equal to the argument of from. The meaning of "$[\![x > 0]\!]$" in the second take-rule will be made precise in Sect. 2. Intuitively, it allows application of that rule only if the instantiation of the variable x is a positive number. 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 if evaluation of the second argument of ins is forbidden since the recursive call to from is then blocked. ◊

As for ordinary rewriting, termination is a fundamental property of context-sensitive rewriting. Since context-sensitive rewriting may result in a terminating rewrite relation where regular rewriting is diverging, proving termination of

[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.

context-sensitive rewriting is quite challenging. For ordinary TRSs, a promising 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 [27]. 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 compared to the ordinary dependency pair method since dependency pairs may be collapsing, which requires strong restrictions on how the method can be applied.

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 involved and cannot be presented due to space constraints. They can be found in the full version [14], which furthermore contains additional techniques not presented in this paper.

After fixing terminology, Sect. 2 recalls and extends the CERSs introduced in [12]. In contrast to [12], 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 the termination prover AProVE [17] is discussed and evaluated in Sect. 6. This evaluation shows that our implementation succeeds in proving termination of a large class of context-sensitive CERSs.

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 2 ($\mathcal{E}$-Extended Rewriting). *Let $\mathcal{R}$ be a TRS and let $\mathcal{E}$ be a set of equations. Then $s \to_{\mathcal{E}\setminus\mathcal{R}} t$ 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 $\xrightarrow{>\Lambda}_{\mathcal{E}\setminus\mathcal{R}}$ denotes that all steps are applied below the root, and $\xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\setminus\mathcal{R}}$ denotes normalization w.r.t. $\xrightarrow{>\Lambda}_{\mathcal{E}\setminus\mathcal{R}}$.

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 $\mathcal{F}_{\mathbb{Z}} = \{\mathsf{0} : \to \mathtt{int},\ \mathsf{1} : \to \mathtt{int},\ - : \mathtt{int} \to \mathtt{int},\ + : \mathtt{int} \times \mathtt{int} \to \mathtt{int}\}$. Terms over $\mathcal{F}_{\mathbb{Z}}$ are written using a simplified notation, e.g., $x-2$ instead of $x+((-1)+(-1))$.

As is well-known, *equational completion* [21, 6] generates the following rules $\mathcal{S}_{\mathbb{Z}}$ and equations $\mathcal{E}_{\mathbb{Z}}$ from the defining properties of Abelian groups:

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

Recall that equality w.r.t. the properties of Abelian groups is reduced to $\mathcal{E}_{\mathbb{Z}}$-equivalence of $\to_{\mathcal{E}_{\mathbb{Z}}\setminus\mathcal{S}_{\mathbb{Z}}}$-normal forms. This idea can be used for natural numbers with $\mathcal{F}_{\mathbb{N}} = \{\mathsf{0} : \to \mathtt{nat},\ \mathsf{1} : \to \mathtt{nat},\ + : \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat}\}$, $\mathcal{S}_{\mathbb{N}} = \{x + \mathsf{0} \to x\}$, and $\mathcal{E}_{\mathbb{N}} = \{x+y \approx y+x,\ x+(y+z) \approx (x+y)+z\}$ as well [12]. In the following, $\mathcal{T}h$ denotes one of $\mathbb{Z}$ or $\mathbb{N}$, and $\mathtt{base}$ denotes the sort $\mathtt{int}$ or $\mathtt{nat}$, 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 defined functions then have constraints over these predicate symbols that guard when a rewrite step may be performed. To this end, an *atomic $\mathcal{T}h$-constraint* has the form $t_1\ P\ t_2$ for a predicate symbol $P \in \mathsf{P}$ and terms $t_1, t_2 \in \mathcal{T}(\mathcal{F}_{\mathcal{T}h}, \mathcal{V})$. The set of *$\mathcal{T}h$-constraints* is the closure of the set of atomic $\mathcal{T}h$-constraints under $\top$ (truth), $\neg$ (negation), and $\wedge$ (conjunction). The Boolean connectives $\vee$, $\Rightarrow$, and $\Leftrightarrow$ can be defined as usual. Also, $\mathcal{T}h$-constraints have the expected semantics. The main interest is in *$\mathcal{T}h$-satisfiability* (i.e., the constraint is true for some instantiation of its variables) and *$\mathcal{T}h$-validity* (i.e., the constraint is true for all instantiations of its variables). Notice that both of these properties are decidable.

In order to extend $\mathcal{F}_{\mathcal{T}h}$ by collection data structures and defined functions, a finite signature $\mathcal{F}$ over the sort `base` 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 `base` 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 the table below. In the following, a combination of $\mathcal{T}h$ with (signature-disjoint) collection data structures $\mathcal{C}_1, \ldots, \mathcal{C}_n$ is considered. To this end, let $\mathcal{S} = \mathcal{S}_{\mathcal{T}h} \cup \bigcup_{i=1}^{n} \mathcal{S}_{\mathcal{C}_i}$ and $\mathcal{E} = \mathcal{E}_{\mathcal{T}h} \cup \bigcup_{i=1}^{n} \mathcal{E}_{\mathcal{C}_i}$.

	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	$\emptyset, \{\cdot\}, \cup$	$x \cup \emptyset \to x$ $x \cup x \to x$ $(x \cup x) \cup y \to x \cup y$ $x \cup (y \cup z) \approx (x \cup y) \cup z$ $x \cup y \approx y \cup x$

The defined functions are specified using constrained rewrite rules. Here, the $\mathcal{T}h$-constraint guards when a rewrite step may be performed.

Definition 3 (Constrained Rewrite Rules). *A* constrained rewrite rule *has the form $l \to r[\![\varphi]\!]$ for terms $l, r \in \mathcal{T}(\mathcal{F} \cup \mathcal{F}_{\mathcal{T}h}, \mathcal{V})$ and a $\mathcal{T}h$-constraint φ such that* $\mathrm{root}(l) \in \mathcal{F} - \mathcal{F}(\mathcal{E} \cup \mathcal{S})$ *and* $\mathcal{V}(r) \cup \mathcal{V}(\varphi) \subseteq \mathcal{V}(l)$.

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 $\mathcal{T}h$

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]. Notice that checking the instantiated constraint for validity requires the matching substitution to be $\mathcal{T}h$-based, i.e., all variables of sort `base` are mapped to terms from $\mathcal{T}(\mathcal{F}_{\mathcal{T}h}, \mathcal{V})$.

Definition 4 (Rewrite Relation of a CERS). *For a CERS $(\mathcal{R}, \mathcal{S}, \mathcal{E})$, let $s \xrightarrow{\mathcal{S}}_{\mathcal{T}h \| \mathcal{E} \setminus \mathcal{R}} t$ iff there exist $l \to r[\![\varphi]\!] \in \mathcal{R}$, a position $p \in \mathcal{P}os(s)$, and a $\mathcal{T}h$-based σ such that (i) $s|_p \xrightarrow{\geq \Lambda\, !}_{\mathcal{E} \setminus \mathcal{S}} \circ \overset{\geq \Lambda}{\sim}_{\mathcal{E}} l\sigma$, (ii) $\varphi\sigma$ is $\mathcal{T}h$-valid, and (iii) $t = s[r\sigma]_p$.*

It is shown in [14] that $\xrightarrow{\mathcal{S}}_{\mathcal{T}h \| \mathcal{E} \setminus \mathcal{R}}$ 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* $\mathcal{D}(\mathcal{R})$.

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 $f \in \mathcal{F} \cup \mathcal{F}_{\mathcal{T}h}$. 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 $\mathcal{P}os^{\mu\!\!\!/}(t) = \mathcal{P}os(t) - \mathcal{P}os^{\mu}(t)$. 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, $t \triangleright_{\mu\!\!\!/} s$ 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, $\mathcal{V}^{\mu\!\!\!/}(t)$ 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:

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

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

The rewrite relation of a CS-CERS is obtained by a small modification of Def. 4 such that the position where the reduction takes place has to be active.

Definition 5 (Rewriting with a CS-CERS). *For a CS-CERS* $(\mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$*, let* $s \xrightarrow{\mathcal{S}}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu} t$ *iff there exist* $l \to r[\![\varphi]\!] \in \mathcal{R}$*, an active position* $p \in \mathcal{P}os^{\mu}(s)$*, and a* $\mathcal{T}h$*-based substitution* σ *such that (i)* $s|_p \xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\setminus\mathcal{S}} \circ \overset{\geq\Lambda}{\sim}_{\mathcal{E}} l\sigma$*, (ii)* $\varphi\sigma$ *is* $\mathcal{T}h$*-valid, and (iii)* $t = s[r\sigma]_p$*.*

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

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

Notice that an infinite reduction of this term 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. $\Diamond$

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. To this end, 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 $\mathcal{F}$. 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 $t^{\sharp}$. Then 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 $\mathrm{root}(t) \in \mathcal{D}(\mathcal{R})$. 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 $l \not\trianglerighteq_{\mu} x$ for some rule $l \to r$. The reason for this is that recursive calls occuring 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 since it is hard to extend methods for proving termination from ordinary rewriting to context-sensitive rewriting. While progress has been made [2, 3, 19], 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 only consider instantiations by *hidden terms*, which are terms with a defined root symbol occurring inactively in right-hand sides of rules.

Definition 7 (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 $r \rhd_\mu t$.*

In Ex. 6, the term $\mathsf{from}(x{+}1)$ is hidden since $\mathsf{ins}(x,\mathsf{from}(x{+}1)) \rhd_\mu \mathsf{from}(x{+}1)$. As shown in [1] for ordinary TRSs, it does not suffice to only consider 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 differs from the one given in [1] by also considering $\mathcal{S}$ and $\mathcal{E}$.

Definition 8 (Hiding Contexts). *Given a CS-CERS $(\mathcal{R},\mathcal{S},\mathcal{E},\mu)$, $f \in \mathcal{F} \cup \mathcal{F}_{\mathcal{T}h}$ 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 $r \rhd_\mu s$ and $s_i \trianglerighteq_\mu x$ for an $x \in \mathcal{V}$ or $s_i \trianglerighteq_\mu g(\ldots)$ 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. 6, $+$ 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 $\mathsf{U}_{\mathtt{base}}$ and $\mathsf{U}_{\mathtt{univ}}$ and *unhiding* dependency pairs are used.

Definition 9 (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*

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

Here, $\mathsf{U}_{\mathtt{base}} : \mathtt{base} \to \mathtt{top}$ 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, $\mu(\mathsf{U}_{\mathtt{base}}) = \mu(\mathsf{U}_{\mathtt{univ}}) = \emptyset$ and $\mu(f^\sharp) = \mu(f)$ for all $f \in \mathcal{F}$.

Example 10. For Ex. 6, $\mathsf{DP}(\mathcal{R},\mu)$ is as follows:

$$\begin{aligned}
\mathsf{take}^\sharp(x,\mathsf{ins}(y,ys)) &\to \mathsf{take}^\sharp(x-1,ys)\ [\![x>0]\!] && (1)\\
\mathsf{take}^\sharp(x,\mathsf{ins}(y,ys)) &\to \mathsf{U}_{\mathtt{base}}(y)\ [\![x>0]\!] && (2)\\
\mathsf{take}^\sharp(x,\mathsf{ins}(y,ys)) &\to \mathsf{U}_{\mathtt{univ}}(ys)\ [\![x>0]\!] && (3)\\
\mathsf{pick}^\sharp(\mathsf{ins}(x,xs)) &\to \mathsf{U}_{\mathtt{base}}(x) && (4)\\
\mathsf{drop}^\sharp(\mathsf{ins}(x,xs)) &\to \mathsf{U}_{\mathtt{univ}}(ys) && (5)\\
\mathsf{U}_{\mathtt{univ}}(\mathsf{from}(x+1)) &\to \mathsf{from}^\sharp(x+1) && (6)\\
\mathsf{U}_{\mathtt{base}}(x+y) &\to \mathsf{U}_{\mathtt{base}}(x) && (7)\\
\mathsf{U}_{\mathtt{base}}(x+y) &\to \mathsf{U}_{\mathtt{base}}(y) && (8)\\
\mathsf{U}_{\mathtt{base}}(-x) &\to \mathsf{U}_{\mathtt{base}}(x) && (9)\\
\mathsf{U}_{\mathtt{univ}}(\mathsf{from}(x)) &\to \mathsf{U}_{\mathtt{base}}(x) && (10)
\end{aligned}$$

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 $\xrightarrow{\mathcal{S}}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu}$ is terminating if there are no infinite minimal chains.

Definition 11 ((Minimal) $(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$-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 $\mathcal{T}h$-based substitution σ such that $t_i\sigma \xrightarrow{\mathcal{S}}{}^{*}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu} \circ \xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\setminus\mathcal{S}} \circ \overset{\geq\Lambda}{\sim}_{\mathcal{E}} s_{i+1}\sigma$ and $\varphi_i\sigma$ is $\mathcal{T}h$-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 $\xrightarrow{\mathcal{S}}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu}$-reduction for all $i \geq 1$.*

Here, $\xrightarrow{\mathcal{S}}{}^{*}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu}$ corresponds to reductions occurring strictly below the root of $t_i\sigma$ and $\xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\setminus\mathcal{S}} \circ \overset{\geq\Lambda}{\sim}_{\mathcal{E}}$ 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 main result for CS-CERSs constitutes the main technical contribution of this paper. The proof requires several technical lemmas and can be found in [14].

Theorem 12. *Let $(\mathcal{R},\mathcal{S},\mathcal{E},\mu)$ be a CS-CERS. Then $\xrightarrow{\mathcal{S}}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu}$ 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 [18]. 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.

5 CS-DP Processors

This section introduces several sound CS-DP processors. Most of these processors are similar to corresponding processors developed for the non-context-sensitive case in [12]. The soundness proofs for the CS-DP processors are, however, more involved than the corresponding soundness proofs in [12], see [14] for details.

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 13 (Estimated Context-Sensitive Dependency Graphs). *For a CS-DP problem $(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$, the nodes in 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 $\mathrm{CAP}_\mu(t_1)\sigma \xrightarrow{>\Lambda}{}^{!}_{\mathcal{E}\setminus\mathcal{S}} \circ \overset{\geq\Lambda}{\sim}_{\mathcal{E}} s_2\sigma$ and $\varphi_1\sigma$, $\varphi_2\sigma$ are $\mathcal{T}h$-valid. Here, CAP_μ is given by*

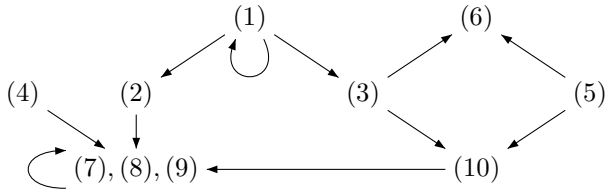
1. *for $x \in \mathcal{V}$, $\mathrm{CAP}_\mu(x) = x$ if $\mathrm{sort}(x) = \mathsf{base}$ and $\mathrm{CAP}_\mu(x) = y$ otherwise,*
2. *$\mathrm{CAP}_\mu(f(t_1,\ldots,t_n)) = f(t'_1,\ldots,t'_n)$ if $f \notin \mathcal{D}(\mathcal{R})$, where $t'_i = t_i$ if $i \notin \mu(f)$ and $t'_i = \mathrm{CAP}_\mu(t_i)$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

In both cases, 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]. The following CS-DP processor uses the estimated dependency graph in order to decompose a CS-DP problem into several independent CS-DP problems by considering the *strongly connected components (SCCs)* of $\mathsf{EDG}(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$.

Theorem 14 (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 SCCs of $\mathsf{EDG}(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$, is sound.*

Example 15. For the dependency pairs from Ex. 10, the following estimated dependency graph $\mathsf{EDG}(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)$ is obtained:



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. 14, the following CS-DP problems are obtained:

$$(\{(1)\},\mathcal{R},\mathcal{S},\mathcal{E},\mu) \qquad (11) \qquad\qquad (\{(7),(8),(9)\},\mathcal{R},\mathcal{S},\mathcal{E},\mu) \qquad (12)$$

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

5.2 Subterm Criterion

The subterm criterion for ordinary TRSs [20] 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 its direct

subterms. Given a set $\mathcal{P}$ of dependency pairs and a subset $\mathcal{P}' \subseteq \mathcal{P}$, the method consists of finding a projection such that the collapsed right-hand side is a subterm of the collapsed left-hand side for all dependency pairs in $\mathcal{P}$, where this subterm relation is furthermore strict for all dependency pairs from $\mathcal{P}'$. Then the dependency pairs in $\mathcal{P}'$ may be removed from the CS-DP problem.

Definition 16 (Projections). *A* projection *is a mapping π that assigns to every $f^\sharp \in \mathcal{F}^\sharp$ with $\mathrm{arity}(f^\sharp) = n$ 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.*

In the context of CERSs, the subterm relation modulo $\mathcal{E}$ can be used [12]. 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 17 ($\mathcal{E}$-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 $s \trianglerighteq_{\mathcal{E},\mu} t$, iff $s \rhd_{\mathcal{E},\mu} t$ or $s \sim_\mathcal{E} t$.*

It is shown in [14] that $\rhd_{\mathcal{E},\mu}$ and $\trianglerighteq_{\mathcal{E},\mu}$ are decidable for the sets $\mathcal{E}$ of equations considered in this paper. Furthermore, the following properties can be shown.

Lemma 18. *Let $(\mathcal{R}, \mathcal{S}, \mathcal{E}, \mu)$ be a CS-CERS. Then $\rhd_{\mathcal{E},\mu}$ is well-founded and $\rhd_{\mathcal{E},\mu}$ and $\trianglerighteq_{\mathcal{E},\mu}$ are stable and compatible with $\sim_\mathcal{E}$.*

Now the subterm criterion as outlined above can easily be implemented using a 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 19 (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*
  - *$\pi(s) \trianglerighteq_{\mathcal{E},\mu} \pi(t)$ 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 20. Recall the CS-DP problem (12) from Ex. 15, consisting of (7)–(9). Using $\pi(\mathsf{U}_{\mathsf{base}}) = 1$, this CS-DP problem can easily be handled. ◊

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. The idea for this is simple: If all dependency pairs from a CS-DP problem are at least weakly decreasing, then all dependency pairs that are strictly decreasing cannot occur infinitely often in infinite chains and may thus be deleted.

Often, *reduction pairs* [22] 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.[3]

[3] It is also possible to develop an abstract 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

- 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 $\mathcal{P}ol(+) = x_1 + x_2$,
- 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 $\mathrm{arity}(f) = n$, and
- 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 $\mathrm{arity}(f^\sharp) = n$ 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 21 ($\succ_{\mathcal{P}ol}$, $\succsim_{\mathcal{P}ol}$, 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, $s \succsim_{\mathcal{P}ol} t$ 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 22 ($\succ_{\mathcal{P}ol}$ and $\succsim_{\mathcal{P}ol}$ 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 $s[\![\varphi]\!] \succsim_{\mathcal{P}ol} t[\![\varphi]\!]$ iff $s\sigma \succsim_{\mathcal{P}ol} t\sigma$ 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:

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

Here, $x_1, \ldots, x_n$ are the variables occurring in $[s]_{\mathcal{P}ol}$ or $[t]_{\mathcal{P}ol}$. This requirement might be impossible to show if one of the x_i has sort `univ` since then the possible values it can take are not restricted by the $\mathbb{Z}$-constraint φ. By restricting the $\mathbb{Z}$-polynomial interpretation $\mathcal{P}ol$ such that for each $f \in \mathcal{F}$ with resulting sort `univ`, the polynomial $\mathcal{P}ol(f)$ may only depend on a variable x_i if the i^{th} argument of f has sort `univ`, an easier requirement is obtained since ground terms of sort `univ` are then mapped to non-negative integers. Thus, it suffices to show that

$$\forall x_1, \ldots, x_k.\ \forall y_1 \geq 0, \ldots, y_l \geq 0.\ \varphi \Rightarrow [s]_{\mathcal{P}ol} \geq c_{\mathcal{P}ol}$$
$$\forall x_1, \ldots, x_k.\ \forall y_1 \geq 0, \ldots, y_l \geq 0.\ \varphi \Rightarrow [s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$$

are true in the integers, where $x_1, \ldots, x_k$ are the variables of sort `base` in $[s]_{\mathcal{P}ol}$ or $[t]_{\mathcal{P}ol}$ and $y_1, \ldots, y_l$ are the variables of sort `univ` in $[s]_{\mathcal{P}ol}$ or $[t]_{\mathcal{P}ol}$.

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

$\xrightarrow{\mathcal{S}}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu}$ takes place below an occurrence of $-$. There are two possibilities:

1. All arguments of right-hand sides of $\mathcal{P}$ are terms from $\mathcal{T}(\mathcal{F}_{\mathbb{Z}},\mathcal{V})$. Then no reductions w.r.t. $\xrightarrow{\mathcal{S}}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu}$ can take place between instantiated dependency pairs in a chain since chains are built using $\mathbb{Z}$-based substitutions.
2. $\mathcal{F}$ does not contain a function symbol with resulting sort `base`. Then, terms with sort `univ` do not occur below occurrences of $-$ and only terms with sort `univ` are reducible by $\xrightarrow{\mathcal{S}}_{\mathcal{T}h\|\mathcal{E}\setminus\mathcal{R},\mu}$.

These possibilities give rise to two CS-DP processors. For the first possibility, notice that $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{E}$ do not need to be considered.

Theorem 23 (CS-DP Processor Using $\mathbb{Z}$-Polynomial Interpretations–Version 1). *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* $s\to t[\![\varphi]\!]\in\mathcal{P}'$
  - $s[\![\varphi]\!] \gtrsim_{\mathcal{P}ol} t[\![\varphi]\!]$ *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* [22] for the function symbols $f^\sharp\in\mathcal{F}^\sharp$ that ensures that condition *1* is true afterwards.

Example 24. Recall the CS-DP problem (11) from Ex. 15, 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

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

Now condition *1* from above is satisfied and $\mathbb{Z}$-polynomial interpretations are applicable. Indeed, using $\mathcal{P}ol(\mathsf{take}^\sharp)=x_1$ concludes the termination proof of the running example. ◊

If condition *2* from above is satisfied, the following CS-DP processor can be obtained. A refinement that only needs to consider syntactically determined subsets of $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{E}$ is presented in [14].

Theorem 25 (CS-DP Processor Using $\mathbb{Z}$-Polynomial Interpretations–Version 2). *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* $\mathcal{F}$ *does not contain a function symbol with resulting sort* `base`, $\mathcal{P}ol$ *is a* $\mathbb{Z}$*-polynomial interpretation,* $\mathcal{P}'\subseteq\mathcal{P}$, *and*
  - $s[\![\varphi]\!] \succ_{\mathcal{P}ol} t[\![\varphi]\!]$ *for all* $s\to t[\![\varphi]\!]\in\mathcal{P}'$
  - $s[\![\varphi]\!] \gtrsim_{\mathcal{P}ol} t[\![\varphi]\!]$ *for all* $s\to t[\![\varphi]\!]\in(\mathcal{P}-\mathcal{P}')\cup\mathcal{R}$
  - $l \gtrsim_{\mathcal{P}ol} r$ *for all* $l\to r\in\mathcal{S}$
  - $u \sim_{\mathcal{P}ol} v$ *for all* $u\approx v\in\mathcal{E}$
- $\{(\mathcal{P},\mathcal{R},\mathcal{S},\mathcal{E},\mu)\}$, *otherwise.*

Then Proc *is sound.*

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 as part of the termination prover AProVE [17], resulting in AProVE-CERS. While most of the implementation is relatively straightforward, the automatic generation of suitable $\mathbb{Z}$-polynomial interpretations is non-trivial. Details on this can be found in [13, 10]. 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, e.g., 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. Within 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 about 2 seconds for one example. 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). 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, usually by one or two orders of magnitude (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 is available at `http://www.cs.unm.edu/~spf/tdps/`.

References

1. B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving context-sensitive dependency pairs. In *LPAR '08*, LNAI 5330, pages 636–651, 2008.
2. B. Alarcón, R. Gutiérrez, and S. Lucas. Context-sensitive dependency pairs. In *FSTTCS '06*, LNCS 4337, pages 297–308, 2006.

3. B. Alarcón, R. Gutiérrez, and S. Lucas. Improving the context-sensitive dependency graph. *ENTCS*, 188:91–103, 2007.
4. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1–2):133–178, 2000.
5. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
6. L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. *TCS*, 67(2–3):173–201, 1989.
7. F. Blanqui, T. Hardin, and P. Weis. On the implementation of construction functions for non-free concrete data types. In *ESOP '07*, LNCS 4421, pages 95–109, 2007.
8. C. Borralleras, S. Lucas, and A. Rubio. Recursive path orderings can be context-sensitive. In *CADE '02*, LNAI 2392, pages 314–331, 2002.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*. LNCS 4350. Springer-Verlag, 2007.
10. S. Falke. *Term Rewriting with Built-In Numbers and Collection Data Structures*. PhD thesis, University of New Mexico, Albuquerque, NM, USA, 2009. To appear.
11. S. Falke and D. Kapur. Dependency pairs for rewriting with non-free constructors. In *CADE '07*, LNAI 4603, pages 426–442, 2007.
12. S. Falke and D. Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In *RTA '08*, LNCS 5117, pages 94–109, 2008.
13. S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *CADE '09*, LNAI 5663, pages 277–293, 2009.
14. S. Falke and D. Kapur. Termination of context-sensitive rewriting with built-in numbers and collection data structures. Technical Report TR-CS-2009-01, 2009.
15. M. C. F. Ferreira and A. L. Ribeiro. Context-sensitive AC-rewriting. In *RTA '99*, LNCS 1631, pages 286–300, 1999.
16. C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *RTA '09*, LNCS 5595, pages 32–47, 2009.
17. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2. In *IJCAR '06*, LNAI 4130, pages 281–286, 2006.
18. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework. In *LPAR '04*, LNAI 3452, pages 301–331, 2005.
19. R. Gutiérrez, S. Lucas, and X. Urbain. Usable rules for context-sensitive rewrite systems. In *RTA '08*, LNCS 5117, pages 126–141, 2008.
20. N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *IC*, 205(4):474–511, 2007.
21. J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194, 1986.
22. K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *PPDP '99*, LNCS 1702, pages 47–61, 1999.
23. S. Lucas. Context-sensitive computations in functional and functional logic programs. *JFLP*, 1998(1), 1998.
24. S. Lucas. Termination of rewriting with strategy annotations. In *LPAR '01*, LNAI 2250, pages 669–684, 2001.
25. S. Lucas. Context-sensitive rewriting strategies. *IC*, 178(1):294–343, 2002.
26. S. Lucas. Lazy rewriting and context-sensitive rewriting. *ENTCS*, 64:234–254, 2002.
27. S. Lucas. Polynomials for proving termination of context-sensitive rewriting. In *FOSSACS '04*, LNCS 2987, pages 318–332, 2004.

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 method of 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 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. 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 often use 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 property is important, so we need some way to ensure termination of rewrite rules. 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]. The format CRS is known as the first detailed formulation of higher-order rewriting system (i.e. rewriting system having the feature of *variable binding* and meta-level substitutions) in the theory of term rewriting. A CRS is essentially a set of rewrite rules on second-order terms. In this paper, we give a method to prove termination (meaning strong normalisation) of a CRS, called *higher-order semantic labelling*. This is an extension of semantic labelling for first-order term rewriting systems (TRSs) given by Zantema [Zan95]. Let us look at examples first.

Example 1 (CRS for 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]).

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

We try to prove termination of the CRS $\mathcal{P}$. A powerful decidable method to prove termination is known for CRSs, called *General Schema* [Bla00]. The idea of 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 (as in the primitive recursion) extended in a multiset or lexicographic manner. However, General Schema *cannot* show termination of the CRS $\mathcal{P}$ 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. Intuitively, we know that the map function does not change the length of the argument of a list, thus we can see that a shorter list than $x : xs$ is used in the recursive call of ps. To prove termination of ps, this "semantic" information (rather than syntactical structural decreasingness) must be taken into account.

Higher-order semantic labelling developed in this paper is a method to reflect such information in rewrite rules. Here we use the "length" of a list for ps as the semantics. By using higher-order semantic labelling, we obtain 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 $i \in \mathbb{N}$ (cf. Sec. 4.4). This i denotes the length of the argument of ps. Then, General Schema succeeds in showing termination of the labelled rules with the precedence $\mathsf{ps}_i > \mathsf{ps}_j > \mathsf{map} > :$ for $i > j \in \mathbb{N}$ because this time the subterm comparison is not needed by using $\mathsf{ps}_{i+1} > \mathsf{ps}_i$. The main theorem (Thm. 9) of higher-order semantic labelling we obtain is that if the labelled CRS is terminating, then the original CRS is terminating. Hence, we can conclude termination of $\mathcal{P}$.

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 showed usefulness of higher-order semantic labelling by several examples for which General Schema alone fails.

Background. Higher-order semantic labelling was firstly introduced for Inductive Datatype Systems [Ham07]. This paper simplifies the labelling method and applies it to examples taken from functional programming. The semantics used in this paper is based on the algebraic semantics of CRS [Ham05], 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 a further application of this Σ-monoid structure. The outline of higher-order semantic labelling for CRSs (without proofs) has appeared in

13th International Conference on Logic for Programming Artificial Intelligence Reasoning (LPAR'06) as a short paper.

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. For lack of space, 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:

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

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 no z in l and r, the substitute $\theta(\mathrm{z})$ has 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 only CRSs built from binding signatures (cf. Aczel's contraction schemes [Acz78]), which we call *structural CRSs*. A *(binding) signature* Σ is consisting of a set Σ of function symbols with an arity function $a : \Sigma \to \mathbb{N}^*$ ($\mathbb{N}^*$ 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 ($1 \leq i \leq l$). For a formal treatment of named variables modulo α-equivalence in CRSs, we assume the method of de Bruijn levels [FPT99] 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 $t ::= x \mid f(x_1 \cdots x_{i_1}.t_1, \ldots, x_1 \cdots x_{i_r}.t_l) \mid \mathrm{z}^l[t_1, \ldots, t_l]$ 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 another part is not important.

3 Semantics of CRSs

3.1 Binding Algebras

We review the notion of binding algebras [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 $\Sigma A = \coprod_{f:\langle n_1,\ldots,n_l\rangle \in \Sigma} \prod_{1 \leq i \leq l} \delta^{n_i} A$. 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 a *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.

An intuition of a Σ-monoid is an algebra with the operation of an interpretation of *substitution* on (semantics of) terms. This semantic substitution operation is called *multiplication*, typically denoted by β and μ in this paper. Why this is a multiplication is that the substitution operation satisfies the monoid law in an abstract setting.

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 $n{+}\vec{k}.t$. 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 $(t_1, \ldots, t_l) \longmapsto f(n{+}\vec{i_1}.t_1, \ldots, n{+}\vec{i_l}.t_l)$. 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.

$$\begin{aligned} \beta(n)(\underline{i}; \vec{t}) &= t_i \qquad \beta(n)(\mathrm{z}[s_1, \ldots, s_l]; \vec{t}) = \mathrm{z}[\beta(n)(s_1; \vec{t}), \ldots, \beta(n)(s_l; \vec{t})] \\ \beta(n)(f(m{+}\vec{i_1}.s_1, \ldots, m{+}\vec{i_l}.s_l); \vec{t}) &= f(m{+}\vec{i_1}.\beta(m{+}i_1)(s_1; \mathsf{up}_{i_1}(\vec{t}), m{+}1, \ldots, m{+}i_1), \ldots \\ &\qquad m{+}\vec{i_l}.\beta(m{+}i_l)(s_l; \mathsf{up}_{i_l}(\vec{t}), m{+}1, \ldots, m{+}i_l) \end{aligned}$$

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 we abuse the notation to use Z to denote its presheaf version $\hat{Z} \in \mathbf{Set}^{\mathbb{F}}$ in an assignment. We use the following "homomorphic extension" in this paper.

Definition 2. 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

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

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 just say "a CRS" for a structural CRS.

The notions of models and quasi-models for CRSs can be defined by immediate variations of algebraic semantics in [Ham05]. 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 $\mathrm{V}+\Sigma$-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 extend semantic labelling for TRSs by Zantema [Zan95] by a more 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 *sort 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 $f : \langle i_1, \ldots, i_l\rangle \in \Sigma$, a *sort 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 sort 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 3 (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

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

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

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

We state the following characterisation that clarifies what is the mathematical structure of semantic labelled meta-terms. While it is mathematically natural, this algebraic viewpoint was not considered in the original proposal [Zan95].

Theorem 4. *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 5. *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 $(M_\Sigma Z, [f_T]_{f\in\Sigma}, \nu, \beta) \to (M_{\overline{\Sigma}}Z, [f_\phi]_{f\in\Sigma}, \nu_\phi, \beta_\phi)$.*

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

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

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

$$\begin{aligned}
&\beta_\phi(n)(x;\, \vec{t}) = t_x \\
&\beta_\phi(n)(\mathsf{z}[s_1, \ldots, s_l];\, \vec{t}) = \mathsf{z}[\beta_\phi(n)(s_1; \vec{t}), \ldots, \beta_\phi(n)(s_l; \vec{t})] \\
&\beta_\phi(n)(f_q(m{+}\vec{i_1}.s_1, \ldots, m{+}\vec{i_l}.s_l);\, \vec{t}) \\
&\quad = \begin{cases} f_p(m{+}\vec{i_1}.\beta_\phi(m{+}i_1)(s_1;\, \mathsf{up}_{m+i_1}(\vec{t}), m{+}1, \ldots, m{+}i_1), \ldots) & \text{if } m+1 > n \\ f_p(n{+}\vec{i_1}.\beta_\phi(n{+}i_1)(s_1;\, \mathsf{up}_{n+i_1}(\vec{t}), n{+}1, \ldots, n{+}i_1), \ldots) & \text{if } m+1 \le n \end{cases}
\end{aligned}$$

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 6), to perform (essentially) the β-reduction of an instantiated meta-application, such as an instance of $\mathrm{F}[\mathrm{x}]$
- substitution of metavariables (written as θ in Lemma 7), used to instantiate rewrite rules, and formally called valuation (Def. 2).

The labelling map ϕ^{L} has to commute with these two substitutions. Why 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. 8). This process requires to "push" substitutions outside of an application of the labelling map in a term in two levels (i.e. for variables and for metavariables). This is nothing but commutativity of labelling with substitutions.

Lemma 6. *Let $\phi : 0 \longrightarrow M$ be an assignment. Then, the following diagram commutes in $\boldsymbol{Set}^{\mathbb{F}}$:*

$$\begin{array}{ccc} M_\Sigma 0 \bullet M_\Sigma 0 & \xrightarrow{\beta} & M_\Sigma 0 \\ {\scriptstyle \phi^{\mathsf{L}} \bullet \phi^{\mathsf{L}}}\downarrow & & \downarrow{\scriptstyle \phi^{\mathsf{L}}} \\ M_{\overline{\Sigma}} 0 \bullet M_{\overline{\Sigma}} 0 & \xrightarrow[\beta_\phi]{} & M_{\overline{\Sigma}} 0 \end{array}$$

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

Lemma 7. *Let $\phi : 0 \longrightarrow M$ and $\theta : Z \longrightarrow M_\Sigma 0$ be assignments. Then, the following diagram commutes in $\boldsymbol{Set}^{\mathbb{F}}$:*

$$\begin{array}{ccc} M_\Sigma Z & \xrightarrow{\theta^*} & M_\Sigma 0 \\ {\scriptstyle (\phi^*\theta)^{\mathsf{L}}}\downarrow & & \downarrow{\scriptstyle \phi^{\mathsf{L}}} \\ M_{\overline{\Sigma}} Z & \xrightarrow[(\phi^{\mathsf{L}}\theta)^{\overline{*}}]{} & M_{\overline{\Sigma}} 0 \end{array}$$

Here $(-)^{\overline{*}}$ denotes the $\overline{\Sigma}$-monoid morphism extension $(-)^*$ (cf. Def. 2) 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

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

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 8. *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 $n \vdash \phi_n^{\mathsf{L}} s \to_{\overline{\mathcal{R}}} \phi_n^{\mathsf{L}} t$ on $M_{\overline{\Sigma}} 0_n$.*

Theorem 9 (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. 8. [$\Rightarrow$]: 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 General Schema for CRSs [Bla00] to prove termination of labelled CRSs because in our experience, this is the most powerful decidablemethod to prove termination of CRSs. General Schema uses a *precedence* which is a partial order on function symbols occurring in a CRS. Using a precedence, if General Schema satisfies all rewrite rules of a given CRSs, we conclude termination of it [Bla00].

Example 10 (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 $\Sigma = \{\mathsf{map} : \langle 1, 0\rangle, \mathsf{S}, \mathsf{ps} : \langle 0\rangle,\ \mathsf{0}, \mathsf{nil} : \langle\rangle, +, \text{“}:\text{”} : \langle 0, 0\rangle\}$.

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 (cf. [FPT99] Sec. 2) in the monoidal category $(\mathbf{Set}^{\mathbb{F}}, \bullet, \mathrm{V})$ 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 sort set S_{ps}. The sort map is defined by $\langle\!\langle x \rangle\!\rangle_0^{\mathsf{ps}} = x$. Then, we have the following labelled rules

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

for all $i \in \mathbb{N}$. 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 sort set S_f was a set. Here we require them to be equipped with well-founded partial orders. The operations $f_{\mathcal{M}}$ and sort 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 *sorts* and a *sort 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 sort set S_f as in Sec. 4. Let $(M, \geq_M)$ be a quasi-model for a CRS $\mathcal{R}$. Using the sort 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 11. *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 $\phi : 0 \longrightarrow M$, $n \vdash \phi_n^{\mathsf{L}} s \to_{\mathsf{Decr}}^* ; \to_{\overline{\mathcal{R}}} \phi_n^{\mathsf{L}} t$ holds. Here ";" denotes the sequential composition of relations.*

Theorem 12. *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.*

Example 13 (CRS for quick sort). We define the CRS $\mathcal{R}$ for quick sort by the following rules [BR01] with the rewrite rules of standard functions: if,$+$, filter, ">", "≤".

$$\begin{array}{llll} 0 > \mathrm{y} & \to \mathsf{false} & 0 \leq \mathrm{y} & \to \mathsf{true} \\ \mathrm{x} > 0 & \to \mathsf{true} & \mathsf{s}(\mathrm{x}) \leq 0 & \to \mathsf{false} \\ \mathsf{s}(\mathrm{x}) > \mathsf{s}(\mathrm{y}) & \to \mathrm{x} > \mathrm{y} & \mathsf{s}(\mathrm{x}) \leq \mathsf{s}(\mathrm{y}) & \to \mathrm{x} \leq \mathrm{y} \\ \mathsf{if}(\mathsf{true}, \mathrm{x}, \mathrm{y}) & \to \mathrm{x} & \mathsf{nil} +\!\!+ \mathrm{xs} & \to \mathrm{xs} \\ \mathsf{if}(\mathsf{false}, \mathrm{x}, \mathrm{y}) & \to \mathrm{y} & (\mathrm{x} : \mathrm{xs}) +\!\!+ \mathrm{ys} & \to \mathrm{x} : (\mathrm{xs} +\!\!+ \mathrm{ys}) \end{array}$$

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

To show termination of this CRS $\mathcal{R}$ by employing an RPO-like method is again difficult because 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 ($\mathrm{x} : \mathrm{xs}$). Thus, General Schema fails to show termination. The higher-order RPO 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. We use the same carrier $\mathcal{M}_n \triangleq (\mathbb{N}^n \to \mathbb{N})$ as in Example 10, equipped with the usual order $\ge$ on $\mathbb{N}$ and its pointwise extension. We give a quasi-model that estimates the maximum bounds of the lengths of lists appearing in the arguments of functions in $\mathcal{R}$. The operations $\mathsf{nil}_{\mathcal{M}_0}$ and $:_{\mathcal{M}_0}$ are the same as in Example 10 and other operations are:

$$
\begin{gathered}
\mathsf{qsort}_{\mathcal{M}_0}(x) = x \qquad \mathsf{filter}_{\mathcal{M}_0}(p, x) = x \\
+\!\!+_{\mathcal{M}_0}(x, y) = \mathsf{max}(x, y) \qquad :_{\mathcal{M}_0}(x, y) = y + 1 \qquad \mathsf{if}_{\mathcal{M}_0}(x, y, z) = \mathsf{max}(y, z)
\end{gathered}
$$

where max is the maximum function on $\mathbb{N}$. This is indeed a quasi-model and cannot be a model because the interpretation of the rule for qsort (the last rule) is decreasing ($\ge$). We label the function symbol qsort only. The sort map is the same as the case of ps: $\langle\!\langle - \rangle\!\rangle^{\mathsf{qsort}} = \langle\!\langle - \rangle\!\rangle^{\mathsf{ps}}$, which is weakly monotone. Then, we have the labelled rules:

$$
\begin{aligned}
\mathsf{qsort}_0(\mathsf{nil}) &\to \mathsf{nil} \\
\mathsf{qsort}_{i+1}(\mathrm{x} : \mathrm{xs}) &\to \mathsf{qsort}_i(\mathsf{filter}(a.a \le \mathrm{x}, \mathrm{xs})) +\!\!+ ((\mathrm{x} : \mathsf{nil}) +\!\!+ \\
&\qquad \mathsf{qsort}_i(\mathsf{filter}(a.a > \mathrm{x}, \mathrm{xs}))) && \text{for all } i \in \mathbb{N} \\
\mathsf{qsort}_i(\mathrm{xs}) &\to \mathsf{qsort}_j(\mathrm{xs}) && \text{for all } i > j \in \mathbb{N}
\end{aligned}
$$

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

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] P. Aczel. A general Church-Rosser theorem. Technical report, University of Manchester, 1978.

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

[Bla00] Frederic Blanqui. Termination and confluence of higher-order rewrite systems. In *Rewriting Techniques and Application (RTA 2000)*, LNCS 1833, pages 47–61. Springer, 2000.

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

[BR01] Cristina Borralleras and Albert Rubio. A monotonic higher-order semantic path ordering. In *Procs. 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, LNCS 2250, pages 531–547, 2001.

[DR98] O. Danvy and K.H. Rose. Higher-order rewriting and partial evaluation. In *Rewriting Techniques and Applications, 9th International Conference, (RTA'98)*, LNCS 1379, 1998.

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

[Ham04] M. Hamana. Free Σ-monoids: A higher-order syntax with metavariables. In *Asian Symposium on Programming Languages and Systems (APLAS 2004)*, LNCS 3302, pages 348–363, 2004.

[Ham05] M. Hamana. Universal algebra for termination of higher-order rewriting,. In *Proceedings of 16th International Conference on Rewriting Techniques and Applications (RTA'05)*, LNCS 3467, pages 135–149. Springer, 2005.

[Ham07] Makoto Hamana. Higher-order semantic labelling for inductive datatype systems. In *Ninth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 97–108, 2007.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, Amsterdam, 1980. volume 127 of Mathematical Centre Tracts.

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

[Mac71] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1971.

[Ter03] Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

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

A Appendix

A.1 Proof of Corollary 5

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 $z \mapsto z$. It is clear that $i_\phi^* = \phi^\sqcup$ by just comparing the definitions of $\phi^\sqcup$ and the Σ-monoid extension $(-)^*$. Hence $\phi^\sqcup$ gives a Σ-monoid morphism.

A.2 Proof of Lemma 7

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.

$$\begin{aligned}
\mathrm{lhs} &= \phi^{\mathsf{L}}\theta^*(\mathrm{z}[\vec{t}]) \\
&= \phi^{\mathsf{L}}\beta(\theta\mathrm{z}; \theta^*\vec{t}) = \beta_\phi(\phi^{\mathsf{L}}\theta\mathrm{z}; \phi^{\mathsf{L}}\theta^*\vec{t}) \qquad \text{(by Lemma 6)} \\
\mathrm{rhs} &= (\phi^{\mathsf{L}}\theta)^{\overline{*}}(\phi^*\theta)^{\mathsf{L}}\mathrm{z}[\vec{t}] \\
&= (\phi^{\mathsf{L}}\theta)^{\overline{*}}\mathrm{z}[(\phi^*\theta)^{\mathsf{L}}\vec{t}] \\
&= \beta_\phi(\phi^{\mathsf{L}}\theta\mathrm{z}; (\phi^{\mathsf{L}}\theta)^{\overline{*}}(\phi^*\theta)^{\mathsf{L}}\vec{t}) \\
&= \beta_\phi(\phi^{\mathsf{L}}\theta\mathrm{z}; \phi^{\mathsf{L}}\theta^*\vec{t}) = \mathrm{lhs} \qquad \text{(by I.H.)}
\end{aligned}$$

A.3 Proof of Proposition 8

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 7 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.4 Proof of Proposition 11

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. 8.

(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 $\phi^{\mathsf{L}}_{n+i}s \to^*_{\mathsf{Decr}}; \to_{\overline{\mathcal{R}}} \phi^{\mathsf{L}}_{n+i}t$. Notice also that $\langle\!\langle - \rangle\!\rangle$ is weakly monotone. So,

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

A.5 Proof of Theorem 12

For both directions, we prove contrapositions. [⇐]: By Prop. 11. [⇒]: By erasing all labels in rewrite steps.

A.6 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.

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., [5, 7, 23, 29] 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 [2], 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., [25, 4, 24, 27, 26]). 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 [3]), etc., which are not useful in our context.

In this work, we consider strong (quasi-)termination[3] so that our results can be applied to any application domain where the selection rule is not known in

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* 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).

[3] 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.

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 [14], 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 limitation of previous approaches to the offline partial evaluation of logic programs like, e.g., [6], 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 [17, 30], we have shown that this drawback can be overcome by using instead a strong termination analysis based on the size-change principle [15, 28]. 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 [6].

The new BTA scheme of [17], however, still had some shortcomings concerning both efficiency and accuracy. In particular, the size-change analysis involves computing the transitive closure of the so-called *size-change graphs* of the program. This is often an expensive process with a worst case exponential growth factor [15].

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.

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

$$p(X) \leftarrow q(X,Y), p(Y).$$
$$q(s(X), X).$$

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 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 [15] was originally aimed at proving the termination of functional programs. This analysis was adapted to the logic programming setting in [30], where both termination and quasi-termination were analysed. The main difference w.r.t. previous termination analyses for logic programs is that [30] 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 scalable binding-time analysis; see [17] 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 [28], in [30] 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 $||\cdot||_{ts}$ [9] 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 $\succsim$.

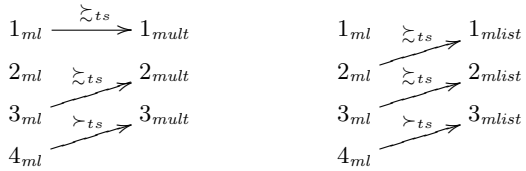
A size-change graph is thus a bipartite labelled graph $\mathcal{G} = (V, W, E)$ where $V = \{1_p, \ldots, n_p\}$ and $W = \{1_q, \ldots, m_q\}$ are the labels of the output and input nodes, respectively, and $E \subseteq V \times W \times \{\succsim, \succ\}$ are the edges.

Example 1. Consider the following program *MLIST*:

(c_1) $mlist(L, I, [\,]) \leftarrow empty(L).$
(c_2) $mlist(L, I, LI) \leftarrow nonempty(L), hd(L, X), tl(L, R), ml(X, R, I, LI).$
(c_3) $ml(X, R, I, [XI|RI]) \leftarrow mult(X, I, XI),\ mlist(R, I, RI).$
(c_4) $mult(0, Y, 0).$ (c_5) $mult(s(X), Y, Z) \leftarrow mult(X, Y, Z1),\ add(Z1, Y, Z).$
(c_6) $add(X, 0, X).$ (c_7) $add(X, s(Y), s(Z)) \leftarrow add(X, Y, Z).$
(c_8) $hd([X|\_], X).$ (c_9) $empty([\,]).$
(c_{10}) $tl([\_|R], R).$ (c_{11}) $nonempty([\_|\_]).$

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:[4]



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 a transitive closure of the size-change graphs by composing them in all possible ways.

Definition 2 (graph concatenation, idempotent multigraph). *A multigraph of P is inductively defined to be either a size-change graph of P or the concatenation (see below) of two multigraphs of P. Given two multigraphs:*

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1) \quad \textit{and} \quad \mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

w.r.t. the same reduction pair $(\succsim, \succ)$, then the concatenation

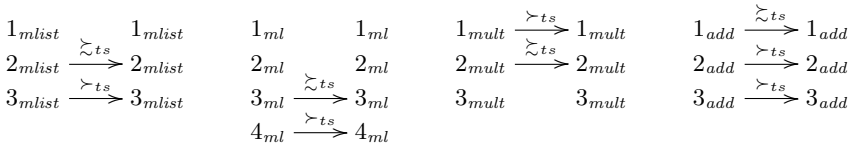
$$\mathcal{G} \bullet \mathcal{H} = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E)$$

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.

[4] 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.

Example 2. For the program *MLIST* of Example 1, we have the following four idempotent multigraphs:



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 [17, 30] 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.[5]
- A predicate p/n quasi-terminates if every idempotent multigraph for p/n contains edges $j_p^1 \xrightarrow{R_1} 1_p, \ldots, j_p^n \xrightarrow{R_n} n_p$, $R_i \in \{\succ, \succsim\}$, 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* [8, 29], 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.
2. Then, after initialising a set $\mathcal{M}$ with the computed size-change graphs, one proceeds iteratively as follows:
   (a) compute the concatenation 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. Intuitively speaking, it improves the naive procedure by taking into account the following observations:

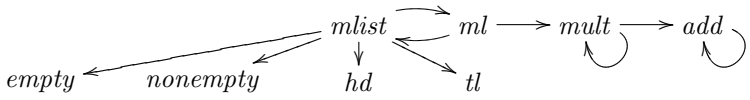
[5] A more relaxed condition based on the notion of *instantiated enough* w.r.t. a norm [22] can be found in [17].

– Firstly, not all size-change graphs need to be constructed, but only those in the path of a (potential) loop. For instance, in Example 1, the size-change graph from *mlist* to *empty* cannot contribute to the construction of any idempotent multigraph.
– Secondly, in many cases, computing the idempotent multigraphs for a single predicate for each loop suffices. 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 $\mathcal{G}_2$.

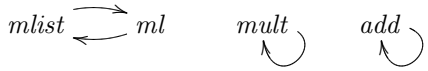
These observations allow us to design a faster procedure for size-change analysis. It proceeds in a stepwise manner as follows:

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[6] $p(\overline{t_n}) \leftarrow B_1, \ldots, q(\overline{s_m}), \ldots, B_k$, $k \geq 1$, in the program.

For instance, the call graph of program *MLIST* in Example 1 is as follows:



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 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:



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*

[6] We use $\overline{t_n}$ to denote the sequence $t_1, \ldots, t_n$.

multigraphs can only be built from the concatenation 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 concatenations 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 Ex. 1, 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 $\succ \not\sqsubseteq \succsim$.

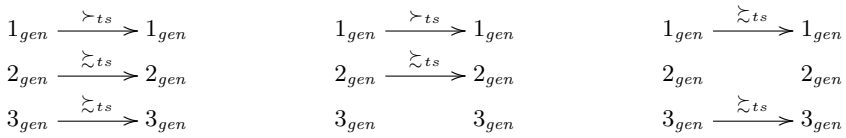
Basically, if a multigraph $\mathcal{G}$ is weaker than another multigraph $\mathcal{H}$, then we have that whenever termination can be proved with $\mathcal{G}$ only, it could also be proved with both $\mathcal{G}$ and $\mathcal{H}$. Indeed, if $\mathcal{G} \sqsubseteq \mathcal{H}$ and $\mathcal{G}' \sqsubseteq \mathcal{H}'$ 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 [18]:

$$\begin{array}{l} gen(or(X,\_),H,T) \leftarrow gen(X,H,T).\\ gen(or(\_,Y),H,T) \leftarrow gen(Y,H,T).\\ gen(star(\_),T,T).\\ gen(star(X),H,T) \leftarrow gen(X,H,T1), gen(star(X),T1,T). \end{array}$$

Here, we have the following three size-change graphs:[7]



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 concatenated with other graphs.

The algorithm of Fig. 1 follows these principles:

- In every iteration, we only consider concatenations 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 concatenations 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 concatenations 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:

[7] 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:**
   $i := 0$; $\mathcal{M}_i := i\_sc\_graphs(P, S)$; $SC := sc\_graphs(P)$
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

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. $\mathcal{M}'$.*

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 [14]. *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.

In the following, *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 two elements: static and dynamic; more refined domains can be found in, e.g., [6].

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—perhaps generalising some of their 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:[8]

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; 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 [17], which includes none of the new ideas presented in this paper.

An interesting implementation technique, which all three versions consider (not described in [17]), is the use of hashing[9] to more quickly identify which size-change graphs already exist and which ones can be concatenated with each

[8] The groundness of an argument is now replaced by the argument being static.

[9] 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.

other. All these three algorithms are integrated into the same BTA from [17], 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.

Evaluation of Efficiency. Figure 2 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 [18] library, vanilla, ctl and lambdaint come from [16]. The picemul program is the PIC processor emulator from [11] with 137 clauses and 855 lines of code. javabc and javabc_heap are Java Bytecode interpreters from [10] with roughly 100 clauses. peval are over 2500 lines of Prolog from a partial evaluator for the ground representation from [20]. 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 [31]. csp is the core interpreter for full CSP-M from [21], consisting of 1771 lines of code. prob is the core interpreter of ProB [19] 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., [13]), consisting of 1148 lines of code. Finally, goedel is the source code of the Gödel system [12] consisting of 27354 lines of Prolog.[10] The "noentry" annotation in Fig. 2 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 [17] 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 inferred 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.

[10] 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.

Benchmark	Old BTA from [17]	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. 2. Empirical results

Compared to the BTA from [6] 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 [17]. We have also tried the latest version of Terminweb,[11] based upon [5]. However, the online version failed to terminate successfully on, e.g., the picemul example (for which our size-change analysis takes 0.01 s). We have also tried TermiLog,[12] 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 SCC optimisations in the algorithm of Fig. 1, the precision remains unchanged w.r.t. [17], and as such the same specialisations can be achieved as described in [17] using hints: e.g., Jones-optimal specialisation for vanilla, reproducing the decompilation from Java bytecode to CLP from [10] or automatically generating the generated code from [11] 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 [17] and now only one hint is required to obtain a good specialisation. For lambdaint 6 hints were

[11] `http://www.cs.bgu.ac.il/~mcodish/TerminWeb/`
[12] `http://www.cs.huji.ac.il/~naomil/termilog.php`

required in [17] 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.[13] In the following section we show how the precision of the size-change 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:

$$(c_2) \quad mlist(L, I, LI) \leftarrow nonempty(L), hd(L, X), tl(L, R), ml(X, R, I, LI).$$

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 never selected. 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., [5], while size relations of success patterns can be obtained from the computation of the convex hull of [1]. 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 induce the reduction pair considered in 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

$$P \leftarrow Q_1, \ldots, Q_{i-1}, p(t_1, \ldots, t_n), Q_{i+1}, \ldots, Q_m.$$

[13] 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`.

together with the hint `'$FULLYUNFOLD'(p,n,I)`. Then, we first replace this clause by the following ones:

$$P \leftarrow Q_1, \ldots, Q_{i-1}, p_{entry}(x_1, \ldots, x_k, t_1, \ldots, t_n).$$
$$p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n) \leftarrow p(y_1, \ldots, y_n), p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n).$$
$$p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n) \leftarrow Q_{i+1}, \ldots, Q_m.$$

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

$$p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_1 \leftarrow p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_1.$$
$$\ldots$$
$$p_{entry}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_j \leftarrow p_{exit}(x_1, \ldots, x_k, y_1, \ldots, y_n)\sigma_j.$$

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 $i_{p_{entry}} \overset{\succ}{\longmapsto} j_{p_{exit}}$ (resp. $i_{p_{entry}} \overset{\succsim}{\longmapsto} j_{p_{exit}}$) to the size-change graph from p_{entry} to p_{exit}. Moreover, we add an edge of the form $i_{p_{entry}} \overset{\succsim}{\longmapsto} i_{p_{exit}}$ 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

$$(c_{21}) \quad mlist(L, I, LI) \leftarrow nonempty(L), hd_{entry}(L, X, I, LI).$$
$$(c_{22}) \quad hd_{entry}(L, X, I, LI) \leftarrow hd(L, X), hd_{exit}(L, X, I, LI).$$
$$(c_{23}) \quad hd_{exit}(L, X, I, LI) \leftarrow tl_{entry}(L, R, X, I, LI).$$
$$(c_{24}) \quad tl_{entry}(L, R, X, I, LI) \leftarrow tl(L, R), tl_{exit}(L, R, X, I, LI).$$
$$(c_{25}) \quad tl_{exit}(L, R, X, I, LI) \leftarrow ml(X, R, I, LI).$$

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}:

$$1_{hd_{entry}} \xrightarrow{\succsim_{ts}} 1_{hd_{exit}} \qquad 1_{tl_{entry}} \xrightarrow{\succsim_{ts}} 1_{tl_{exit}}$$
$$1_{hd_{entry}} \xrightarrow{\succ_{ts}} 2_{hd_{exit}} \qquad 1_{tl_{entry}} \xrightarrow{\succ_{ts}} 2_{tl_{exit}}$$
$$2_{hd_{entry}} \xrightarrow{\succsim_{ts}} 2_{hd_{exit}} \qquad 2_{tl_{entry}} \xrightarrow{\succsim_{ts}} 3_{tl_{exit}}$$
$$3_{hd_{entry}} \xrightarrow{\succsim_{ts}} 3_{hd_{exit}} \qquad 3_{tl_{entry}} \xrightarrow{\succsim_{ts}} 4_{tl_{exit}}$$
$$4_{hd_{entry}} \xrightarrow{\succsim_{ts}} 4_{hd_{exit}} \qquad 4_{tl_{entry}} \xrightarrow{\succsim_{ts}} 5_{tl_{exit}}$$

Finally, by constructing the size-change graphs for clauses c_{21}, c_{23} and c_{25} as usual, the size-change analysis can now infer the right relation between the sizes of list L in the atom $mlist(L, I, LI)$ and the head X and tail R in the atom $ml(X, R, I, LI)$.

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 [19]. Together with the selective use of hints [17], 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 [19] 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. F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver. *TPLP*, 5(1-2):259–271, 2005.
2. M. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.
3. Annalisa Bossi, Sandro Etalle, and Sabina Rossi. Properties of input-consuming derivations. *TPLP*, 2(2):125–154, 2002.
4. Annalisa Bossi, Sandro Etalle, Sabina Rossi, and Jan-Georg Smaus. Termination of simply moded logic programs with dynamic scheduling. *ACM Trans. Comput. Log.*, 5(3):470–507, 2004.
5. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
6. S.-J. Craig, J. Gallagher, M. Leuschel, and K.S. Henriksen. Fully Automatic Binding Time Analysis for Prolog. In *Proc. of LOPSTR'04*, pages 53–68. Springer LNCS 3573, 2005.
7. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.
8. S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K.F. Sagonas. Termination Analysis for Tabled Logic Programming. In *Proc. of LOPSTR'97*, pages 111–127. Springer LNCS 1463, 1998.
9. Allen Van Gelder. Deriving constraints among argument sizes in logic programs. *Ann. Math. Artif. Intell.*, 3(2-4):361–392, 1991.
10. Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Improving the decompilation of Java bytecode to Prolog by partial evaluation. *Electr. Notes Theor. Comput. Sci.*, 190(1):85–101, 2007.
11. K.S. Henriksen and J. Gallagher. Abstract interpretation of pic programs through logic programming. In *SCAM*, pp. 184–196. IEEE Computer Society, 2006.
12. Patricia Hill and John W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

13. Gerard J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
14. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
15. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28:81–92, 2001.
16. M. Leuschel, S.-J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising Interpreters Using Offline Partial Deduction. In *Program Development in Computational Logic*, pages 340–375. Springer LNCS 3049, 2004.
17. M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Proc. of LOPSTR'08*, pages 119–134. Springer LNCS 5438, 2009.
18. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
19. Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
20. Michael Leuschel and Danny De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *The Journal of Logic Programming*, 36(2):149–193, August 1998.
21. Michael Leuschel and Marc Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings ICFEM 2008*, LNCS, pages 278–297. Springer-Verlag, 2008.
22. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proc. of Int'l Conf. on Logic Programming (ICLP'97)*, pages 63–77. The MIT Press, 1997.
23. Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Proving Termination for Logic Programs by the Query-Mapping Pairs Approach. In *Program Development in Computational Logic*, pages 453–498, 2004.
24. Elena Marchiori and Frank Teusink. Termination of Logic Programs with Delay Declarations. *J. Log. Program.*, 39(1-3):95–124, 1999.
25. Lee Naish. Coroutining and the construction of terminating logic programs. *Australian Computer Science Communications*, 15(1):181–190, 1993.
26. Jan-Georg Smaus. Termination of Logic Programs Using Various Dynamic Selection Rules. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 43–57. Springer, 2004.
27. J.-G. Smaus, P.M. Hill, and A. King. Verifying termination and error-freedom of logic programs with `block` declarations. *TPLP*, 1(4):447–486, 2001.
28. R. Thiemann and J. Giesl. The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.
29. S. Verbaeten, K. Sagonas, and D. De Schreye. Termination Proofs for Logic Programs with Tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, 2001.
30. G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In *Proc. of PEPM'07*, pages 51–60. ACM Press, 2007.
31. Bram De Wachter, Alexandre Genon, Thierry Massart, and Cédric Meuter. The formal design of distributed controllers with $_{\mathrm{d}}$sl and Spin. *Formal Asp. Comput.*, 17(2):177–200, 2005.

Advances in Type Systems for Functional Logic Programming *

Francisco J. López-Fraguas
Enrique Martin-Martin
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 [3], for its remarkable simplicity and good properties (decidability, existence of principal types, possibility of type inference). Functional logic languages [11, 7, 6], 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

* 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)

are to provide safety, one should be sure that the adopted system has indeed good 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 [4]; 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 [10]. 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 [12].

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 [5] 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 $snd\ X\ Y \to Y$, $and\ true\ X \to X$, $and\ false\ X \to false$, with the usual types inferred by a classical Damas & Milner algorithm. Then we can write the functions $co\ (snd\ X) \to X$ and $cast\ X \to co\ (snd\ X)$, whose inferred types will be $\forall\alpha.\forall\beta.(\alpha \to \alpha) \to \beta$ and $\forall\alpha.\forall\beta.\alpha \to \beta$ 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 rule of $cast$ 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 [5] 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 [5], 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 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.

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.

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* $Pat \ni t ::= X \mid c\ t_1 \dots t_n\ (n \leq k) \mid f\ t_1 \dots t_n\ (n < k)$, 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 $c \in DC$ and $f \in FS$. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ t_1 \dots c_n$ where $c \in DC^n$, and *Higher order patterns* $HOPat = Pat \setminus FOPat$. Expressions $h\ e_1 \dots e_n$ 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: $FV(let_*\ t = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \setminus var(t))$. 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 $PRule \ni r ::= f\ t_1 \dots t_n \to e\ (n \geq 0)$ 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 $Prog \ni \mathcal{P} ::= \{r_1; \dots; r_n\} (n \geq 0)$.

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 $SType \ni \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid C\ \tau_1 \dots \tau_n\ (C \in \mathcal{TC}^n)$. 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 $FTV(\forall \overline{\alpha_i}.\tau) = FTV(\tau) \setminus \{\overline{\alpha_i}\}$. 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 $\{\overline{\beta_i}\} \cap FTV(\forall\overline{\alpha_i}.\tau) = \emptyset$ [13]. 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 [3] 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. $\mathcal{A}$ [3, 13, 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* $\{\overline{\alpha_i}\} = FTV(\tau) \setminus FTV(\mathcal{A})$. As can be seen, $[\mathrm{LET}_m]$ and $[\mathrm{LET}^h_{pm}]$ 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, $[\mathrm{LET}^X_{pm}]$ and $[\mathrm{LET}_p]$ 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 $\alpha \to \alpha$ (with α a variable not appearing in $\mathcal{A}$) the generalization step will assign both the type-scheme $\forall\alpha.\alpha \to \alpha$, losing the connection between them.

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 $FTV(\tau_i) \not\subseteq FTV(\tau)$.*

The previous definition 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.

$$
\begin{array}{lcl}
[\mathbf{ID}] & \dfrac{}{\mathcal{A} \vdash s : \tau} & \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} \\[2ex]
[\mathbf{APP}] & \dfrac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1 e_2 : \tau} & \\[2ex]
[\Lambda] & \dfrac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau} & \text{if } \{\overline{X_i}\} = var(t) \\[2ex]
[\mathbf{LET}_m] & \dfrac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \mathbf{let}_m\ t = e_1\ \mathbf{in}\ e_2 : \tau_2} & \text{if } \{\overline{X_i}\} = var(t) \\[2ex]
[\mathbf{LET}^X_{pm}] & \dfrac{\mathcal{A} \vdash e_1 : \tau_1 \quad \mathcal{A} \oplus \{X : Gen(\tau_1, \mathcal{A})\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \mathbf{let}_{pm}\ X = e_1\ \mathbf{in}\ e_2 : \tau_2} & \\[2ex]
[\mathbf{LET}^h_{pm}] & \dfrac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash h\ t_1 \ldots t_n : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \mathbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \mathbf{in}\ e_2 : \tau_2} & \text{if } \begin{array}{l} \{\overline{X_i}\} = var(t_1 \ldots t_n) \\ \wedge\ h \in DC \cup FS \end{array} \\[2ex]
[\mathbf{LET}_p] & \dfrac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : Gen(\tau_i, \mathcal{A})}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \mathbf{let}_p\ t = e_1\ \mathbf{in}\ e_2 : \tau_2} & \text{if } \{\overline{X_i}\} = var(t) \\[2ex]
\hline
[\mathbf{P}] & \dfrac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash^{\bullet} e : \tau} & \text{if } critVar_{\mathcal{A}}(e) = \emptyset
\end{array}
$$

Fig. 2. Rules of type system

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 $FTV(\alpha_i \pi_g) \not\subseteq FTV(\tau_g)$.*

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 *co* 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 an opaque variable in $snd\ X$ and it occurs in the body, so it is critical.

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 $bool \to bool \to bool$ 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 $f\ t_1 \ldots t_n \to e$ 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 $co\ (snd\ X) \to X$, 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 $\mathcal{P} \equiv \{f\ true \to true; f\ 2 \to false\}$ with the assumptions $\mathcal{A} \equiv \{f :: \forall\alpha.\alpha \to bool\}$. Besides, this restriction is implicitly considered in [5].

$$
\begin{array}{rl}
TRL(s) = s, & \text{if } s \in DC \cup FS \cup \mathcal{DV} \\
TRL(e_1\ e_2) = & TRL(e_1)\ TRL(e_2) \\
TRL(let_K\ X = e_1\ in\ e_2) = & let_K\ X = TRL(e_1)\ in\ TRL(e_2), \text{ with } K \in \{m, p\} \\
TRL(let_{pm}\ X = e_1\ in\ e_2) = & let_p\ X = TRL(e_1)\ in\ TRL(e_2) \\
TRL(let_m\ t = e_1\ in\ e_2) = & let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) \\
TRL(let_{pm}\ t = e_1\ in\ e_2) = & let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) \\
TRL(let_p\ t = e_1\ in\ e_2) = & let_p\ Y = TRL(e_1)\ in\ \overline{let_p\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2)
\end{array}
$$

for $\{\overline{X_i}\} = var(t) \cap var(e_2)$, $f_{X_i} \in FS^1$ fresh defined by the rule $f_{X_i}\ t \to X_i$, $Y \in \mathcal{DV}$ fresh, t a non variable pattern.

Fig. 3. Transformation rules of let expressions with patterns

For subject reduction to be meaningful, a notion of evaluation is needed. In this paper we consider the *let-rewriting* relation of [10]. 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. 3). There 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. 4). 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) $f\ t_1\theta \ldots t_n\theta \ \rightarrow^l \ r\theta, \quad$ if $(f\ t_1 \ldots t_n \rightarrow r) \in \mathcal{P}$ and $\theta \in \mathcal{PS}ubst$

(LetIn) $e_1\ e_2 \rightarrow^l let_m\ X = e_2\ in\ e_1\ X, \quad$ if e_2 is an active expression, variable application, junk or *let* rooted expression, for X fresh.

(Bind) $let_K\ X = t\ in\ e \ \rightarrow^l \ e[X/t], \quad$ if $t \in Pat$

(Elim) $let_K\ X = e_1\ in\ e_2 \rightarrow^l e_2, \quad$ if $X \notin FV(e_2)$

(Flat$_m$) $let_m\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3 \ \rightarrow^l \ let_K\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3), \quad$ if $Y \notin FV(e_3)$

(Flat$_p$) $let_p\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3 \ \rightarrow^l \ let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(LetAp) $(let_K\ X = e_1\ in\ e_2)\ e_3 \rightarrow^l let_K\ X = e_1\ in\ e_2\ e_3, \quad$ if $X \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \rightarrow^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \rightarrow^l e'$ using any of the previous rules

where $K \in \{m, p\}$

Fig. 4. Higher order *let*-rewriting relation $\rightarrow^l$

Theorem 2 (Type preservation of the let transformation). *Assume $\mathcal{A} \vdash^\bullet e : \tau$ and let $\mathcal{P} \equiv \{\overline{f_{X_i}\ t_i \rightarrow X_i}\}$ be the rules of the projection functions needed in the transformation of e according to Fig. 3. 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 $\mathcal{P} \vdash e \rightarrow^l e'$ 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 establish the types for some symbols in the expression.

In this work we have given the type inference in Fig. 5 a relational style to show the similarities with the typing relation. But in essence, the inference rules represent an algorithm (similar to algorithm $\mathcal{W}$ [3, 13]) which fails if any of the 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.

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 3 shows.

Example 3 (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 $\mathcal{D}_2 \equiv \mathcal{A}[\alpha/int] \vdash^\bullet$

[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]
$$\frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash e_1 e_2 : \alpha\pi | \pi_1\pi_2\pi} \quad if \begin{array}{l} \alpha\ fresh\ type\ variable \\ \wedge\ \pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha) \end{array}$$

[iΛ]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t \Vdash e : \tau | \pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau | \pi_t\pi} \quad if \begin{array}{l} \{\overline{X_i}\} = var(t) \\ \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \end{array}$$

[iLET$_m$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_m\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ \{\overline{X_i}\} = var(t) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iLET$^X_{pm}$]
$$\frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ X = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_1\pi_2}$$

[iLET$^h_{pm}$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash h\ t_1 \ldots t_n : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ h \in DC \cup FS \wedge\ \{\overline{X_i}\} = var(h\ t_1 \ldots t_n) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iLET$_p$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_t\pi_1\pi \oplus \{\overline{X_i : Gen(\alpha_i\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_p\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ \{\overline{X_i}\} = var(t) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iP]
$$\frac{\mathcal{A} \Vdash e : \tau | \pi}{\mathcal{A} \Vdash^{\bullet} e : \tau | \pi} \quad if\ critVar_{\mathcal{A}\pi}(e) = \emptyset$$

Fig. 5. Inference rules

$\lambda(snd'\ X).X : (bool \to bool) \to int$. 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).
$\Pi^\bullet_{\mathcal{A},e} = \{\pi \in \mathcal{TS}ubst \mid \exists \tau \in \mathcal{T}ype.\ \mathcal{A}\pi \vdash^\bullet e : \tau\}$

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* $\iff \exists \tau_g, \pi_g \in \mathcal{T}ype.\mathcal{A} \Vdash^\bullet e : \tau_g|\pi_g$.
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, 9]. 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* $\mathbf{pair} : \forall\alpha.\alpha \to \alpha \to \alpha$.

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. 3.

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* $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.

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 4 (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 [5] treats with technical detail a type system for functional logic programming. Our paper makes clear contributions when compared to [5]:

- 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* [8], 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 [5]. 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 [5] wrt. a narrowing calculus. Here we do it wrt. an small-step operational semantics closer to real computations.
- In [5] 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 $\mathcal{TOY}$ [11] 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 [10], and taking into account known problems [5, 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. S. Antoy and A. P. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Proc. International Symposium on Functional and Logic Programming (FLOPS 1999)*, pages 335–353, 1999.

2. B. Brassel. Post to the curry mailing list. http://www.informatik.uni-kiel.de/ curry/listarchive/0706.html, May 2008.
3. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. Symposium on Principles of Programming Languages (POPL 1982)*, pages 207–212, 1982.
4. J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP 1997)*, pages 153–167. MIT Press, 1997.
5. J. González-Moreno, T. Hortalá-González, and Rodríguez-Artalejo, M. Polymorphic types in functional logic programming. In *Journal of Functional and Logic Programming*, volume 2001/S01, pages 1–71, 2001.
6. M. Hanus. Multi-paradigm declarative languages. In *Proc. of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
7. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
8. S. L. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 50–61. ACM, 2006.
9. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993.
10. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
11. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
12. F. J. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Advances in type systems for functional-logic programming (extended version). http://gpd.sip.ucm.es/enrique/papers/ATSFLPlong.pdf, 2009.
13. L. M. Martins Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985. Also appeared as Technical report CST-33-85.
14. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.
15. J. J. Moreno-Navarro, J. Mariño, A. del Pozo-Pietro, Á. Herranz-Nieva, and J. García-Martín. Adding type classes to functional-logic languages. In *1996 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'96*, pages 427–438, 1996.
16. A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Colloquium on International Symposium on Programming*, pages 217–228, London, UK, 1984. Springer-Verlag.
17. S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
18. B. P. Pierce. *Advanced topics in types and programming languages*. MIT Press, Cambridge, MA, USA, 2005.
19. C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.
20. R. Sedgewick. *Algorithms in C++, Part 5: Graph Algorithms*, pages 205–216. Addison-Wesley Professional, 2002.

A Complete Axiomatization of Strict Equality over Infinite Trees*

Javier Álvez and Francisco J. López-Fraguas

Universidad Complutense de Madrid
javieralvez@fdi.ucm.es fraguas@sip.ucm.es

Abstract. Computing with data values that are some kind of trees — finite, infinite, rational— is at the core of declarative programming, either logic, functional or functional-logic. Understanding the logic of trees is therefore a fundamental question with impact in different aspects, like language design, including constraint systems or constructive negation, or obtaining methods for verifying and reasoning about programs. The theory of *true* equality over finite or infinite trees is quite well known. In particular, a seminal paper by Maher proved its decidability and gave a complete axiomatization of the theory. However, the sensible notion of equality for functional and functional-logic languages with a lazy evaluation regime is *strict* equality, a computable approximation to true equality for possibly infinite and partial trees. In this paper, we investigate the first-order theory of strict equality, arriving to remarkable and not obvious results: the theory is again decidable and admits a complete axiomatization, not requiring predicate symbols other than strict equality itself. Besides, the results stem from an effective —taking into account the intrinsic complexity of the problem— decision procedure that can be seen as a constraint solver for general strict equality constraints.

1 Introduction

Computing with data values that are —or can be interpreted as— some kind of trees is at the core of declarative programming, either logic, functional or functional-logic programming. The family of trees may vary from finite trees, for the case of standard logic programming, infinite rational trees, for the case of Prolog-II and variants, or infinite trees (that correspond to data values in constructor data-types) for the case of functional or functional-logic programming that allow non-terminating programs by following a lazy evaluation regime.

Understanding trees, in particular the logical principles governing tree equality, is a fundamental question with impact in different aspects of declarative programming languages. For instance, adding constructive negation abilities to a logic language requires solving complex Herbrand constraints over finite trees.

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407, UCM-BSCH-GR58/08-910502, TIN2007-66523 and GIU07/35.

The theory of *true* equality $\approx$[1] over finite or infinite trees is quite well known. In a seminal paper [11], Maher proved its decidability and gave a complete axiomatization for the cases of finite and infinite trees, and finite and infinite signatures. In another influential paper [6], the authors provided more effective decision procedures, based on reduction to solved forms by quantifier elimination.

However, true equality is not the sensible notion to consider in lazy functional or functional-logic languages like Haskell, Curry or Toy [12, 8, 4]. In those, the possibility of non-terminating programs handled with lazy evaluation implies that denotations of expressions may be seen as infinite trees. Furthermore, trees may be *partial*, in the sense that some of their components may be undefined. For instance, with the definitions $loop = loop$ and $l = [loop|l]$, l can be seen as computing the infinite partial list $[\bot, \bot, \bot, \ldots]$. True equality over partial trees is not Scott continuous (hence not computable) and, therefore, must be replaced by a computable approximation, like *strict* equality $==$, the restriction of $\approx$ to finite and total trees. The theories of equality and of strict equality are far from being the same, starting by the fact that $==$ is not even reflexive.

As far as we know, a comprehensive study of the full first-order theory of strict equality has not been done before. Certainly, strict equality is incorporated as primitive in the aforementioned languages, and there are several works incorporating various Herbrand constraint systems —and corresponding solving procedures— to functional logic languages [2, 9, 3]. But in all cases, the considered class of formulae over $==$ is only a subset of general first-order formulae.

The aim of this paper is precisely to investigate the full first-order theory of strict equality over the algebra $\mathcal{IT}$ of possibly infinite partial trees. Note that decidability and existence of complete axiomatization for $\approx$ says nothing about the same problems for strict equality, even if $==$ is a strict subset of $\approx$ (i.e., $\forall x \cdot y \; (\; x == y \rightarrow x \approx y \;)$ is valid in $\mathcal{IT}$). These are indeed the main questions tackled in this paper:

- Does the theory of strict equality over $\mathcal{IT}$ admit a complete recursive axiomatization?
- In the affirmative case, is it possible that the axioms use only the symbol $==$? We cannot discard *a priori* the possibility that the axiomatization needs an explicit connection of $==$ to $\approx$, like the one stated above. If so, the resulting axiomatization would become more complicated due to the number of required axioms and transformation rules.
- A complete recursive axiomatization of a theory implies its decidability (at least a brute force decision procedure exists). Can we give a more practical decision procedure, in the style of [6]? As a matter of fact, such a procedure —if existing— will be itself a proof of completeness for the theory.

We obtain affirmative answers to these questions, both in the cases of infinite and finite signatures. Our paper does not look for immediate applications, keeping in a theoretical realm and trying to achieve fundamental and not obvious results about strict equality that could be a basis for potential applications: the design

[1] By *true* equality we mean $t_1 \approx t_2$ iff t_1 and t_2 are the same tree.

of constraint systems more expressive than existing ones or the development of reasoning frameworks for functional-logic programs with built-in equality.

The organization of the paper is as follows: in the next section, we provide preliminary definitions and notation. In Section 3, we give an axiomatization for strict equality and introduce the transformation rules that will be used in the decision methods of Section 4, distinguishing the cases of infinite and finite signatures. Finally, in Section 5, we discuss complexity issues and future work. For the sake of space, proofs have been omitted, but the interested reader may refer to [1].

2 Preliminaries

Let $\mathcal{V}$ be a set of countable variables and $\Sigma = \mathcal{P}_\Sigma \cup \mathcal{F}_\Sigma$ a signature of predicate and function symbols where each symbol s has an associated arity n, denoted by $s_{/n}$, and $\mathcal{P}_\Sigma$ exclusively consists of the symbol $==_{/2}$, known as *strict equality*. For technical convenience, we assume that $\mathcal{F}_\Sigma$ contains at least a 0-ary function symbol (constant), an n-ary function symbol with $n > 0$ and a distinguished 0-ary function symbol $\bot$ known as *bottom*. If Σ contains a finite number of function symbols, then Σ is said to be *finite*. Otherwise, Σ is infinite. By using the name *function*, we follow the tradition of first-order logic, but note that the notion of function corresponds to the notion of free constructor in functional/functional-logic programming and not to defined function, which plays no role in this paper.

We consider the classical definitions of finite and infinite ground trees. The interested reader is referred to [5] for an exhaustive definition. A tree is said to be *partial* if it contains $\bot$ at some node. Otherwise, the tree is *total*. The algebra of finite and infinite trees are respectively denoted by $\mathcal{FT}$ and $\mathcal{IT}$. Besides, we also refer to [6] for the definitions that do not appear in this paper.

A *term* (or *constructor term*) is either a variable $v \in \mathcal{V}$ or an expression of the form $f(t_1, \ldots, t_n)$ where $f_{/n} \in \mathcal{F}_\Sigma$ and $t_1, \ldots, t_n$ are terms. For any terms t and s, the expression $t[s]$ denotes that s occurs in t (that is, s is a *subterm* of t). For any $n > 0$, a n-tuple of terms is denoted by $\langle t_1, \ldots, t_n \rangle$ and abbreviated by $\overline{t}$. When convenient, we also treat $\overline{t}$ as the set of its components. As for the case of trees, a term t is said to be *partial* if $t = s[\bot]$, and t is *total* otherwise. We denote by $\mathtt{Var}(t)$ the set of variables occurring in t. Besides, a term is said to be *ground* iff it is variable-free. The *size* of a term t is the number of function symbols occurring in t.

A *sentence* ϕ is an arbitrary first-order formula built with Σ. In our case, the only predicate symbol is $==$. Thus *atomic formulas* are $true$, $false$, *strict equations* $t_1 == t_2$ or *negated equations* $\neg t_1 == t_2$. Being $\overline{r} = \langle r_1, \ldots, r_n \rangle$ and $\overline{s} = \langle s_1, \ldots, s_n \rangle$, the expression $\overline{r} == \overline{s}$ abbreviates $r_1 == s_1 \wedge \ldots \wedge r_n == s_n$ and $\neg\overline{r} == \overline{s}$ abbreviates the disjunction of negated equations $\neg r_1 == s_1 \vee \ldots \vee \neg r_n == s_n$. Sentences may use propositional connectives $(\neg, \wedge, \vee, \rightarrow, \leftrightarrow)$ and quantifiers $(\exists, \forall)$. The symbol Q stands for both kinds of quantifiers. The set $\mathtt{Free}(\phi)$ of *free* variables of ϕ is defined as usual. If $\mathtt{Free}(\phi) = \emptyset$, ϕ is *closed*.

ϕ^Q denotes the Q-closure of ϕ, and $\phi^{Q\setminus\overline{w}}$ denotes the sentence $Q\overline{v}\ \phi$, where $\overline{v} = \mathtt{Free}(\phi) \setminus \overline{w}$.

Now we recall some semantics of first order logic. An *interpretation* $\mathcal{A}$ is a carrier set A together with interpretations for the symbols in Σ. Given $\mathcal{A}$, an *assignment* σ maps variables to values in A. $\mathcal{A}$ *models* ϕ, written $\mathcal{A} \models \phi$, if $\phi\sigma$ is true (according to standard rules for truth-valuation) in $\mathcal{A}$ for any assignment σ (σ is irrelevant if ϕ is closed).

A *theory* $\mathcal{T}$ is a set of closed sentences. $\mathcal{A}$ is a *model* of $\mathcal{T}$, written $\mathcal{A} \models \mathcal{T}$, if $\mathcal{A} \models \phi$ for each $\phi \in \mathcal{T}$. A formula ϕ is a *logical consequence* of $\mathcal{T}$, written $\mathcal{T} \models \phi$, if $\mathcal{A} \models \phi$ whenever $\mathcal{A} \models \mathcal{T}$. This notation extends naturally to sets Φ of formulas. A sentence ϕ is *satisfiable* (or *solvable*) in $\mathcal{T}$, if $\mathcal{T} \models \phi^{\exists}$. A theory $\mathcal{T}$ is *complete* iff for any closed sentence ϕ either $\mathcal{T} \models \phi$ or $\mathcal{T} \models \neg\phi$ holds. The *theory* $\mathcal{T}_\mathcal{A}$ of $\mathcal{A}$ is the set of all closed ϕ such that $\mathcal{A} \models \phi$. Note that $\mathcal{T}_\mathcal{A}$ is always complete. A complete axiomatization of $\mathcal{A}$ is a theory $\mathcal{S} \subseteq \mathcal{T}_\mathcal{A}$ such that $\mathcal{S} \models \mathcal{T}_\mathcal{A}$ (or, equivalently, $\mathcal{S}$ is a complete theory and $\mathcal{A} \models \mathcal{S}$). Usually one is interested in *recursive* axiomatizations where the property '$\phi \in \mathcal{S}$' is decidable.

Given two sentences ϕ_1 and ϕ_2, a *transformation rule* $\phi_1 \mapsto \phi_2$ replaces any occurrence of ϕ_1 in a formula (module variable renaming) with ϕ_2. The application of a transformation rule R to ϕ_1 yielding ϕ_2 is denoted by $\phi_1 \overset{R}{\rightsquigarrow} \phi_2$. A transformation rule R is said to be *correct* in a theory $\mathcal{T}$ iff for any two formulas ϕ_1 and ϕ_2 such that $\phi_1 \overset{R}{\rightsquigarrow} \phi_2$ we have that ϕ_1 and ϕ_2 are logically equivalent in $\mathcal{T}$, i.e., $\mathcal{T} \models \phi_1 \leftrightarrow \phi_2$.

3 Strict Equality

Strict equality is a particular case of classical equality where, besides being syntactically equal, two terms have to be finite and total to be strictly equal.

Definition 1 (Strict equality). *Two trees t_1 and t_2 are* strictly equal*, denoted by $t_1 == t_2$, iff t_1 and t_2 are the same finite and total tree.* □

Strict equality allows us to characterize the subset of $\mathcal{IT}$ consisting of finite and total trees: x is a finite and total tree $\iff x == x$.

In Figure 1, we propose an axiomatization of infinite trees with strict equality, which is similar, but not equal, to the one of finite trees with equality given in [11]. The main difference comes from the fact that strict equality is not reflexive: non-finite/non-total trees are not strictly equal to themselves. Due to this property, $\bot$ and the remaining function symbols in $\mathcal{F}_\Sigma$ have a different treatment. The strict equality theory of infinite trees consisting of $\mathbf{A_1} - \mathbf{A_6}$ is denoted by $\mathcal{E}$. It is easy to see, by direct inspection, that $\mathcal{IT} \models \mathcal{E}$. However, whether or not $\mathcal{E}$ is a complete theory (and, therefore, a complete axiomatization of $==$ for $\mathcal{IT}$) is not a trivial question. It will be proved by means of a decision procedure based on some equivalences under $\mathcal{E}$ used as transformation rules for quantifier elimination. The following property, which can be seen as a weak version of reflexivity, is logical consequence of $\mathcal{E}$.

Proposition 1. $\mathcal{E} \models \forall x\ (\ x == x\ \leftrightarrow\ \exists y\ x == y\ )$ □

$(\mathbf{A_1})$ For every $f \in \Sigma$ such that $f \neq \bot$

$$\forall \overline{x}\ \forall \overline{y}\ (\ f(\overline{x}) == f(\overline{y})\ \leftrightarrow\ \overline{x} == \overline{y}\ )$$

$(\mathbf{A_2})$ For every $f, g \in \Sigma$ such that $f \neq g$

$$\forall \overline{x}\ \forall \overline{y}\ \neg f(\overline{x}) == g(\overline{y})$$

$(\mathbf{A_3})$ For every term $t[x]$ except x such that $\overline{y} = \mathtt{Var}(t[x]) \setminus \{x\}$

$$\forall x\ \forall \overline{y}\ \neg x == t[x]$$

$(\mathbf{A_4})$

$$\forall x\ \neg x == \bot$$

$(\mathbf{A_5})$ Symmetry

$$\forall x\ \forall y\ (\ x == y\ \rightarrow\ y == x\ )$$

$(\mathbf{A_6})$ Transitivity

$$\forall x\ \forall y\ \forall z\ (\ \neg x == y\ \vee\ \neg y == z\ \vee\ x == z\ )$$

Fig. 1. Axiomatization of Infinite Trees with Strict Equality

3.1 Transformation Rules

In Figure 2, we summarize the transformation rules that are used in Section 4 for providing a decision method for $\mathcal{E}$. Note that some conditions in the rules, like the ones in rule **R**, are not necessary for correctness. Instead, those conditions serve to discard the application of some rules when there exist more suitable ones. Next, we prove that the transformation rules in Figure 2 are correct.

Theorem 1. *The transformation rules in Figure 2 are correct in* $\mathcal{E}$. □

4 A Decision Method for Strict Equality

In this section, we prove that the theory of strict equality is decidable by providing an algorithm that transforms any initial constraint into an equivalent disjunction of formulas in solved form. This algorithm is based on the well-known technique of quantifier elimination, as the algorithms proposed in [6, 11] for the equality theory. As in the above cited works, we distinguish two cases depending on whether the signature is finite or infinite. In the case of infinite signatures, $\mathcal{E}$ is already a decidable theory. However, for dealing with finite signatures, we have to adapt the *Domain Closure Axiom* (see [13]) to the case of strict equality in order to obtain a decidable theory. In the next subsections, we first provide a decision algorithm for the case of infinite signatures and then adapt that algorithm for finite ones.

Bottom

$$(\mathbf{B_1})\ \ x == t[\bot] \ \mapsto false$$
$$(\mathbf{B_2})\ \neg x == t[\bot] \mapsto \ true$$

Non-finite trees

$$(\mathbf{NFT_1})\ \neg x == x \wedge \neg r == s[x] \mapsto \neg x == x$$
$$(\mathbf{NFT_2})\ \ \neg x == x \wedge r == s[x] \ \mapsto \ \ false$$
$$(\mathbf{NFT_3})\qquad \forall y\ \neg x == y \qquad \mapsto \neg x == x$$

Finite trees

$$(\mathbf{FT})\ \ x == x \wedge r == s[x] \ \ \mapsto \ \ r == s[x]$$

Decomposition

$$(\mathbf{D_1})\ \ f(r_1, \ldots, r_n) == f(s_1, \ldots, s_n) \ \mapsto \ \ r_1 == s_1 \wedge \ldots \wedge r_n == s_n$$
$$(\mathbf{D_2})\ \neg f(r_1, \ldots, r_n) == f(s_1, \ldots, s_n) \mapsto \neg r_1 == s_1 \vee \ldots \vee \neg r_n == s_n$$

Clash

$$(\mathbf{C_1})\ \ f(r_1, \ldots, r_m) == g(s_1, \ldots, s_n) \ \mapsto false \ \text{ if } f \neq g$$
$$(\mathbf{C_2})\ \neg f(r_1, \ldots, r_m) == g(s_1, \ldots, s_n) \mapsto \ true \ \ \text{if } f \neq g$$

Occur-check

$$(\mathbf{O_1})\ \ x == t[x] \ \mapsto false \ \text{ if } x \neq t[x]$$
$$(\mathbf{O_2})\ \neg x == t[x] \mapsto \ true \ \ \text{if } x \neq t[x]$$

Replacement

$$(\mathbf{R})\ \ x == t \wedge \varphi[x] \ \ \mapsto \ \ x == t \wedge \varphi[x \leftarrow t] \ \text{ if } t \text{ is total and } x \notin \mathtt{Var}(t)$$

Existential quantification elimination

$$(\mathbf{EE_1})\ \ \exists w\ (\ w == w \wedge \varphi\ ) \ \mapsto \qquad \varphi \qquad \text{if } w \notin \mathtt{Var}(\varphi)$$
$$(\mathbf{EE_2})\ \ \exists w\ (\ w == t \wedge \varphi\ ) \ \mapsto \overline{x} == \overline{x} \wedge \varphi \ \text{ if } t \text{ is total, } \overline{x} = \mathtt{Var}(t) \text{ and } w \notin \mathtt{Var}(t) \cup \mathtt{Var}(\varphi)$$
$$(\mathbf{EE_3})\ \exists w\ (\ \neg w == w \wedge \varphi\ ) \mapsto \qquad \varphi \qquad \text{if } w \notin \mathtt{Var}(\varphi)$$

Existential quantification introduction

$$(\mathbf{EI})\ \ r == s[x] \ \ \mapsto \ \ \exists w\ (\ x == w \wedge r == s[x \leftarrow w]\ )$$

Universal quantification elimination

$$(\mathbf{UE})\ \forall y\ (\neg y == t \vee \varphi) \mapsto \neg\overline{x} == \overline{x} \vee \varphi[y \leftarrow t] \ \text{ if } t \text{ is total, } \overline{x} = \mathtt{Var}(t) \text{ and } y \notin \mathtt{Var}(t)$$

Tautology

$$(\mathbf{T})\ \ \varphi \ \ \mapsto \ \ \varphi \wedge (\ x == x \vee \neg x == x\ )$$

Split

$$(\mathbf{S})\ \neg\exists\overline{w}\exists\overline{z}\ (\ x == t[\overline{w}] \wedge \varphi[\overline{w} \cdot \overline{z}]\ ) \mapsto \neg\exists\overline{w}\ (\ x == t[\overline{w}]\ ) \vee \exists\overline{w}(x == t[\overline{w}] \wedge \neg\exists\overline{z}\ \varphi[\overline{w} \cdot \overline{z}]\ )$$

Fig. 2. Transformation Rules

4.1 Infinite Signatures

In order to provide a decision algorithm, we first define a solved form for infinite signatures, called *basic formula*. Then, we show that two basic Boolean operations —conjunction and negation— can be performed on basic formulas. And, finally, we describe the decision algorithm.

Definition 2. *A* basic formula *for the variables $\overline{x}$ is either true, false or a constraint $\exists \overline{w}\ c(\overline{x}, \overline{w})$ such that*

$$c(\overline{x}, \overline{w}) \ = \ \bigwedge_{x_1 \in \overline{x}^1} \neg x_1 == x_1 \ \wedge \ \overline{x}^2 == \overline{t} \ \wedge \ \bigwedge_{w_i \in \overline{w}} \bigwedge_{j=1}^{n_i} (\neg w_i == s_{ij})^{\forall \setminus \overline{w}}$$

where – $\overline{x} = \overline{x}^1 \cup \overline{x}^2$ *and* $\overline{x}^1 \cap \overline{x}^2 = \emptyset$,
– $\overline{w} = \mathtt{Var}(\overline{t})$ *and* $\overline{x} \cap \overline{w} = \emptyset$,
– *if* s_{ij} *is a variable, then* $s_{ij} \in \overline{w}$, *otherwise* s_{ij} *is total,* $w_i \notin \mathtt{Var}(s_{ij})$ *and* $\mathtt{Var}(s_{ij}) \cap \overline{x} = \emptyset$ *for every* $w_i \in \overline{w}$ *and* $1 \leq j \leq n_i$.

A formula is in basic normal form *if it is of the form $Q\overline{y}\ \varphi[\overline{x} \cdot \overline{y}]$ where φ is a disjunction of basic formulas for $\overline{x} \cdot \overline{y}$.* □

Example 1. Let $\{a_{/0}, g_{/1}, f_{/2}\} \subset \mathcal{F}_\Sigma$ and $\overline{x} = \{x_1, x_2, x_3\} \subset \mathcal{V}$. The sentences $\exists \overline{w}\ (\ \neg x_1 == x_1 \wedge x_2 = g(w_1) \wedge x_3 == g(w_2) \wedge \neg w_1 == w_2 \wedge \forall v\ \neg w_2 == f(a, v)\ )$, $(\ \neg x_1 == x_1 \wedge \neg x_2 == x_2 \wedge \neg x_3 == x_3\ )$ and *true* are basic formulas for $\overline{x}$. □

First, we will show that the notion of basic formula is a solved form.

Theorem 2. *Let Σ be an infinite signature. Any basic formula different from $false$ is satisfiable in $\mathcal{E}$.* □

Then, we describe the transformation of any universally quantified disjunction of negated equations into an equivalent disjunction of basic formulas.

Proposition 2. *Any universally quantified disjunction of negated equations*

$$\forall \overline{v}\ (\ \neg w_1 == t_1 \ \vee \ \neg w_2 == t_2 \ \vee \ \ldots \ \vee \ \neg w_n == t_n\ )$$

where $w_i \notin \overline{v}$ for each $1 \leq i \leq n$ can be transformed into an equivalent disjunction of basic formulas for the variables $\overline{x} = \mathtt{Var}(t_1, \ldots, t_n) \setminus \overline{v}$. □

Using the above transformation, we now describe the implementation of the Boolean operations conjunction and negation on basic formulas.

Proposition 3. *A conjunction of disjunctions of basic formulas for $\overline{x}$ can be transformed into an equivalent disjunction of basic formulas for $\overline{x}$.* □

In the next example, we show the transformation of a conjunction of two basic formulas into an equivalent disjunction of basic formulas for the same variables.

Example 2. Let $\{a_{/0}, g_{/1}, f_{/2}\} \subset \mathcal{F}_\Sigma$ and $\overline{x} = \{x_1, x_2, x_3\} \subset \mathcal{V}$. The conjunction of basic formulas for $\overline{x}$ $\varphi_1 = \exists \overline{w}^1\ c_1(\overline{x}, \overline{w}^1) \wedge \exists \overline{w}^2\ c_2(\overline{x}, \overline{w}^2)$ where

$$c_1(\overline{x}, \overline{w}^1) = \neg x_1 == x_1 \wedge x_2 == w_1^1 \wedge x_3 == g(w_2^1) \wedge \forall v\ \neg w_1^1 == f(a, v)$$
$$c_2(\overline{x}, \overline{w}^2) = \neg x_1 == x_1 \wedge \neg x_2 == x_2 \wedge x_3 == f(w_1^2, w_2^2) \wedge \neg w_1^2 == w_2^2$$

is unsatisfiable since $x_2 == w_1^1 \wedge \neg x_2 == x_2$ is reduced to $false$ by $\mathbf{NFT_2}$. On the contrary, the conjunction $\varphi_2 = \exists \overline{w}^1\ c_1(\overline{x}, \overline{w}^1)\ \wedge\ \exists \overline{w}^3\ c_3(\overline{x}, \overline{w}^3)$ where

$$c_3(\overline{x}, \overline{w}^3)\ =\ \neg x_1 == x_1 \wedge x_2 == f(w_1^3, w_2^3) \wedge x_3 == w_1^3 \wedge \forall v\ \neg w_2^3 == g(v)$$

is transformed in the following way. Since $\sigma = \mathtt{mgu}(w_1^1 \cdot g(w_2^1), f(w_1^3, w_2^3) \cdot w_1^3) = \{w_1^1 \leftarrow f(g(w_2^1), w_2^3),\ w_1^3 \leftarrow g(w_2^1)\}$, φ_2 is transformed into

$$\exists w_2^1 \cdot w_2^3\ (\ \neg x_1 == x_1 \wedge x_2 == f(g(w_2^1), w_2^3) \wedge x_3 == g(w_2^1) \wedge \forall v\ \neg f(g(w_2^1), w_2^3) == f(a, v) \wedge \forall v\ \neg w_2^3 == g(v)\ ).$$

Then, the negated equation $\forall v\ \neg f(g(w_2^1), w_2^3) == f(a, v)$ is reduced to $true$ using rules $\mathbf{D_2}$ and $\mathbf{C_2}$. Thus, the resulting basic formula is

$$\exists w_2^1 \cdot w_2^3\ (\neg x_1 == x_1 \wedge x_2 == f(g(w_2^1), w_2^3) \wedge x_3 == g(w_2^1) \wedge \forall v\ \neg w_2^3 == g(v)).\ \square$$

Proposition 4. *A negated disjunction of basic formulas for the variables $\overline{x}$ can be transformed into an equivalent disjunction of basic formulas for $\overline{x}$.* $\square$

Example 3. Let $\{a_{/0}, f_{/2}\} \subset \mathcal{F}_\Sigma$ and $\overline{x} = \{x_1, x_2\} \subset \mathcal{V}$. The negated basic formula for the variables $\overline{x}$

$$\varphi\ =\ \neg\exists w\ (\ \neg x_1 == x_1 \wedge x_2 == f(w, a) \wedge \forall v\ \neg w == f(a, v)\ )$$

is transformed as follows. First, φ is trivially equivalent to

$$(\ x_1 == x_1\ )\ \vee\ \neg\exists w\ (\ x_2 == f(w, a) \wedge \forall v\ \neg w == f(a, v)\ )$$

where $(\ x_1 == x_1\ )$ is transformed into the following basic formulas for $\overline{x}$

$$\exists w_1\ (\ \neg x_2 == x_2 \wedge x_1 == w_1\ )\ \vee\ \exists w_2 \cdot w_3\ (\ x_1 == w_2 \wedge x_2 == w_3\ )$$

using **T**, **EI**, **R** and **FT**. Besides, the remaining subformula is transformed into

$$\neg\exists w\ (\ x_2 == f(w, a)\ )\ \vee\ \exists w\ (\ x_2 == f(w, a) \wedge \neg\forall v\ \neg w == f(a, v)\ )$$

using the rule **S**. The constraint $\neg\exists w\ (\ x_2 == f(w, a)\ )$ is transformed into

$$\begin{array}{l}(\ \neg x_1 == x_1 \wedge \neg x_2 == x_2\ )\ \vee \\ \exists w_4\ (\ \neg x_1 == x_1 \wedge x_2 == w_4 \wedge \forall v\ \neg w_4 == f(v, a)\ )\ \vee \\ \exists w_5\ (\ \neg x_2 == x_2 \wedge x_1 == w_5\ )\ \vee \\ \exists w_6 \cdot w_7\ (\ x_1 == w_6 \wedge x_2 == w_7 \wedge \forall v\ \neg w_7 == f(v, a)\ )\end{array}$$

$$(\mathbf{EE_4})\ \ \exists\overline{w}\ (\ \overline{v} == \overline{v} \wedge \bigwedge_{i=1}^{n} (\neg s_i == t_i)^{\forall\setminus\overline{w}} \wedge \varphi\ )\ \ \mapsto\ \ \varphi$$

if $\overline{w} \cap \mathtt{Var}(\varphi) = \emptyset$, $\overline{v} \subseteq \overline{w}$, $s_i \neq t_i$, $\overline{w} \cap \mathtt{Var}(s_i, t_i) \neq \emptyset$ and either s_i (resp. t_i) is not a variable or $s_i \in \overline{w}$ (resp. $t_i \in \overline{w}$) for each $1 \leq i \leq n$

Fig. 3. Existential Quantification Elimination: Infinite Signatures

using **T**, **EI**, **R** and $\mathbf{NFT_1}$. Finally, $\exists w\ (\ x_2 == f(w,a) \wedge \neg\forall v\ \neg w == f(a,v)\ ) \equiv \exists w\ (\ x_2 == f(w,a) \wedge \exists v\ w == f(a,v)\ )$ is transformed into

$$\exists w_8\ (\ \neg x_1 == x_1 \wedge x_2 == f(f(a,w_8),a)\ )\ \vee$$
$$\exists w_9 \cdot w_{10}\ (\ x_1 == w_9 \wedge x_2 == f(f(a,w_{10}),a)\ )$$

using rules **R**, $\mathbf{EE_2}$ and **FT** on w, and **T** and **EI** on x_1. □

Next, we show that the elimination of the innermost block of quantifiers is correct in $\mathcal{E}$ when it is existential. For this purpose, we introduce the transformation rule in Figure 3, which allows to eliminate existential variables only occurring in a conjunction of (universally quantified) negated equations. In the next proposition, we prove the correctness of $\mathbf{EE_4}$.

Proposition 5. *Let Σ be an infinite signature. The transformation rule* $\mathbf{EE_4}$ *is correct in $\mathcal{E}$.* □

The elimination of the innermost block of existential quantifiers is used in the decision algorithm given in Figure 4.

Theorem 3. *Let Σ be an infinite signature, $\exists\overline{w}\ a(\overline{x}\cdot\overline{y}, \overline{w}\cdot\overline{z})$ a basic formula for the variables $\overline{x}\cdot\overline{y}$ of the form*

$$\exists\overline{w}\cdot\overline{z}\ (\ \bigwedge_{x_1\in\overline{x}^1} \neg x_1 == x_1 \wedge \bigwedge_{y_1\in\overline{y}^1} \neg y_1 == y_1 \wedge \overline{x}^2 == \overline{t} \wedge \overline{y}^2 == \overline{r} \wedge \varphi \wedge \psi\ )$$

where $-$ $\overline{w} = \mathtt{Var}(\overline{t})$ *and* $\overline{z} = \mathtt{Var}(\overline{r}) \setminus \overline{w}$,
$-$ φ *is a finite conjunction of negated equations such that* $\mathtt{Free}(\varphi) \subseteq \overline{w}$,
$-$ $\psi = \bigwedge_{i=1}^{n} (\neg v_i == s_i)^{\forall\setminus\overline{w}\cdot\overline{z}}$ *and* $(v_i \cup \mathtt{Var}(s_i)) \cap \overline{z} \neq \emptyset$ *for* $1 \leq i \leq n$.
The formulas $\exists\overline{y}\ [\ \exists\overline{w}\ a(\overline{x}\cdot\overline{y}, \overline{w}\cdot\overline{z})\ ]$ *and* $\exists\overline{w}\ a'(\overline{x},\overline{w})$ *where*

$$a'(\overline{x},\overline{w}) \ =\ \bigwedge_{x_1\in\overline{x}^1} \neg x_1 == x_1 \wedge \overline{x}^2 == \overline{t} \wedge \varphi$$

are equivalent in $\mathcal{E}$. □

Example 4. Let $\{a_{/0}, g_{/1}, f_{/2}\} \subset \mathcal{F}_\Sigma$. The formulas

$$\exists y\ [\ \exists w_1 \cdot w_2\ (\ x == g(w_1) \wedge y == f(w_2,a) \wedge \neg w_1 == a \wedge$$
$$\neg w_1 == w_2 \wedge \forall v\ \neg w_2 == f(a,v)\ )\ ]$$

and $\exists w_1\ (\ x == g(w_1) \wedge \neg w_1 == a\ )$ are equivalent in $\mathcal{E}$. □

Given any constraint φ with free variables $\overline{x}^0$:

(Step 1) Transform φ into prenex disjunctive normal form:

$$Q_1\overline{x}^1 \ldots Q_n\overline{x}^n \bigvee_{i=1}^{m} \psi_i$$

(Step 2) For each $1 \leq i \leq m$, transform ψ_i into a disjunction of basic formulas for the variables $\overline{x} = \overline{x}^0 \cdot \overline{x}^1 \cdot \ldots \cdot \overline{x}^n$ as follows:

(a) Apply rules $\mathbf{B_1}$, $\mathbf{B_2}$, $\mathbf{NFT_1}$, $\mathbf{NFT_2}$, $\mathbf{NFT_3}$, $\mathbf{FT}$, $\mathbf{D_1}$, $\mathbf{D_2}$, $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{O_1}$ and $\mathbf{O_2}$. When none of the previous rules is applicable, the resulting formula is a disjunction of constraints of the form

$$\psi_i' \;=\; \bigwedge_{j=1}^{o_1} v_j == r_j \wedge \bigwedge_{j=o_1+1}^{o_2} \neg v_j == r_j$$

where v_i is a variable, r_j is total and $v_j \notin \mathtt{Var}(r_j)$ for each $1 \leq j \leq o_2$.

(b) For each conjunct ψ_i' that results from (a) and each variable $x \in \overline{x}$:

- If $x = v_j$ for some $1 \leq j \leq o_1$, then apply $\mathbf{R}$ on x.
- If $x \neq v_k$ for every $1 \leq j \leq o_1$ and $x \in \mathtt{Var}(r_k)$ for some $1 \leq k \leq o_1$, then apply $\mathbf{EI}$ and $\mathbf{R}$ on x.
- If $x \neq v_j$ and $x \notin \mathtt{Var}(r_j)$ for every $1 \leq j \leq o_1$, then apply $\mathbf{T}$ on x and goto (a).

The resulting formula is already in basic normal form:

$$Q_1\overline{x}^1 \ldots Q_n\overline{x}^n \bigvee_{i=1}^{m'} \exists \overline{w}^i \; a_i(\overline{x}, \overline{w}^i)$$

(Step 3) Iteratively eliminate the innermost block of consecutive existential/universal quantifiers $Q_n\overline{x}^n$:

(i) If $Q_n = \exists$, then $Q_1\overline{x}^1 \ldots Q_{n-1}\overline{x}^{n-1} \bigvee_{i=1}^{m'} \exists \overline{x}^n \cdot \overline{w}^i \; a_i(\overline{x}, \overline{w}^i)$ is equivalent to

$$Q_1\overline{x}^1 \ldots Q_{n-1}\overline{x}^{n-1} \bigvee_{i=1}^{m'} \exists \overline{w}'^i \; a_i'(\overline{x}', \overline{w}'^i)$$

where $\overline{x}' = \overline{x}^0 \cdot \ldots \cdot \overline{x}^{n-1}$ (see Theorem 3).

(ii) If $Q_n = \forall$, then use double negation

$$Q_1\overline{x}^1 \ldots \neg\exists\overline{x}^n \neg \bigvee_{i=1}^{m'} \exists \overline{w}^i \; a_i(\overline{x}, \overline{w}^i)$$

and apply (i). Negation on basic formulas is used before and after the elimination of the innermost block of quantifiers.

Fig. 4. A Decision Method for Strict Equality

The algorithm described in Figure 4 is illustrated in the next example. Roughly speaking, we first transform the input constraint into an equivalent formula in basic normal form. Then, we proceed to iteratively eliminate the innermost block of quantifiers $Q_i\overline{x}^i$. By Theorem 3, the elimination of $Q_i\overline{x}^i$ is trivial when Q_i is existential. However, when Q_i is universal, we have to use double negation to turn Q_i into existential. This process requires to negate the matrix of the formula and to transform it into an equivalent disjunction of basic formulas for the same variables before and after the elimination of Q_i.

Example 5. Let $\{a_{/0}, g_{/1}, f_{/2}\} \subset \mathcal{F}_\Sigma$ and $\overline{x} = \{x_1, x_2\} \subset \mathcal{V}$. The constraint

$$\begin{aligned}\forall\overline{x}\ [\ (\ f(x_1,a) == f(g(x_2),x_2) \wedge \neg g(x_2) == g(g(x_1))\ )\ \vee \\ f(x_1,x_2) == f(x_2,x_1)\ ]\end{aligned}$$

is already in prenex disjunctive normal form, thus **Step 1** is not applicable. In **Step 2**, the formula is first transformed into

$$\forall\overline{x}\ [\ (\ x_1 == g(x_2) \wedge x_2 == a \wedge \neg x_2 == g(x_1)\ )\ \vee\ x_1 == x_2\ ]$$

using $\mathbf{D_1}$ and $\mathbf{D_2}$. Next, the formula is transformed into basic normal form

$$\forall\overline{x}\ [\ (\ x_1 == g(a) \wedge x_2 == a\ )\ \vee\ \exists w\ (\ x_1 == w \wedge x_2 == w\ )\ ]$$

using rules **R**, $\mathbf{C_2}$ and **EI**. Next, in **Step 3**, we proceed to eliminate the innermost block of quantifiers, which is universal (case (ii)). Thus, using double negation, we obtain

$$\neg\exists\overline{x}\ [\ \neg(\ x_1 == g(a) \wedge x_2 == a\ )\ \wedge\ \neg\exists w\ (\ x_1 == w \wedge x_2 == w\ )\ ]$$

that is transformed into

$$\begin{aligned}\neg\exists\overline{x}\ [\ (\ \neg x_1 == x_1 \wedge \neg x_2 == x_2\ )\ \vee\ \exists w\ (\ \neg x_1 == x_1 \wedge x_2 == w\ )\ \vee \\ \exists w\ (\ \neg x_2 == x_2 \wedge x_1 == w \wedge \neg w == g(a)\ )\ \vee \\ \exists\overline{w}\ (\ x_1 == w_1 \wedge x_2 == w_2 \wedge \neg w_1 == g(a) \wedge \neg w_1 == w_2\ )\ \vee \\ (\ \neg x_2 == x_2 \wedge x_1 == g(a)\ )\ \vee \\ \exists w\ (\ x_1 == g(a) \wedge x_2 == w \wedge \neg w == a \wedge \neg w == g(a)\ )\ ]\end{aligned}$$

by negation and conjunction of basic formulas. Then, the block of quantifiers $\exists\overline{x}$ can be eliminated and we obtain $\neg\ [\ true\ ]$, which is trivially equivalent to $false$ after simplifying negation. □

4.2 Finite Signatures

As pointed out in [11] with respect to equality, $\mathcal{E}$ is not a decidable theory in the case of finite signatures. Further, note that the normal form provided in Definition 2 is not solved for finite signatures. This arises from the fact that a finite conjunction of universally quantified negated equations on a variable

($\mathbf{A_7}$) Domain Closure Axiom or DCA

$$\forall x \; (\; \neg x == x \; \vee \bigvee_{f \in \mathcal{F}_\Sigma} \exists \overline{w} \; x == f(\overline{w}) \;)$$

Fig. 5. Axiomatization of Infinite Trees with Strict Equality (contd.)

w may be unsatisfiable if only finite and total trees can be assigned to w. For example, being $\mathcal{F}_\Sigma = \{a_{/0}, g_{/1}\}$, we have that the constraint

$$\exists w \; (\; x == w \wedge \neg w == a \wedge \forall v \; \neg w == g(v) \;)$$

is unsatisfiable although $\exists w \; (\; \neg w == a \wedge \forall v \; \neg w == g(v) \;)$ is satisfiable.

Roughly speaking, the question is that all the function symbols of the signature can be used in a constraint. To solve this problem, we adapt the *Domain Closure Axiom* introduced in [13] to the case of strict equality (see Figure 5), which prevents the existence of isolated finite and total trees in the algebra. Note that $\mathbf{A_7}$ does not provide any information about non-finite/non-total trees. The theory that results from the union of $\mathcal{E}$ and $\mathbf{A_7}$ is denoted by $\mathcal{E}^\star$.

Existential Quantification Elimination

$$(\mathbf{EE_5}) \;\; \exists \overline{w} \; (\; \overline{v} == \overline{v} \wedge \bigwedge_{i=1}^{n} \neg s_i == t_i \wedge \varphi \;) \;\; \mapsto \;\; \varphi$$

if $\overline{w} \cap \mathtt{Var}(\varphi) = \emptyset$, $\overline{v} \subseteq \overline{w}$, $s_i \neq t_i$ and $\overline{w} \cap \mathtt{Var}(s_i, t_i) = \emptyset$ for $1 \leq i \leq n$

Explosion

$$(\mathbf{E}) \; \varphi[x] \;\; \mapsto \;\; \varphi[x] \wedge [\; \neg x == x \vee \bigvee_{f \in \mathcal{F}_\Sigma} \exists \overline{w} \; x == f(\overline{w}) \;]$$

Fig. 6. Transformation Rules: Finite Signatures

Next, we show that $\mathcal{E}^\star$ is a decidable theory. For this purpose, we adapt all the definitions and results in Subsection 4.1 to the case of finite signatures. Besides, we add the new transformation rule **E** (see Figure 6) whose correctness directly follows from Axiom $\mathbf{A_7}$. This new rule allows for the elimination of universal quantification. Finally, regarding the elimination of existential quantifiers, we replace the rule $\mathbf{EE_4}$, which is not correct in the case of finite signatures, with $\mathbf{EE_5}$ (see Figure 6). Next, we show that both rules are correct in $\mathcal{E}^\star$.

Proposition 6. *Let Σ be an finite signature. The transformation rule $\mathbf{EE_5}$ is correct in $\mathcal{E}^\star$.* □

Proposition 7. *The rule* Explosion (**E**) *is correct in $\mathcal{E}^\star$.* □

The use of **E** for eliminating universal quantification is necessary because our notion of solved form for finite signatures is free of universal variables.

Definition 3. *A* basic formula *for the variables $\overline{x}$ is either true, false or a constraint $\exists \overline{w}\ c(\overline{x}, \overline{w})$ such that*

$$c(\overline{x}, \overline{w}) \ = \bigwedge_{x_1 \in \overline{x}^1} \neg x_1 == x_1 \ \wedge \ \overline{x}^2 == \overline{t} \ \wedge \bigwedge_{w_i \in \overline{w}} \bigwedge_{j=1}^{n_i} \neg w_i == s_{ij}$$

where $-$ $\overline{x} = \overline{x}^1 \cup \overline{x}^2$ *and* $\overline{x}^1 \cap \overline{x}^2 = \emptyset$,
$-$ $\overline{w} = \mathtt{Var}(\overline{t})$ *and* $\overline{x} \cap \overline{w} = \emptyset$,
$-$ *if* s_{ij} *is a variable, then* $s_{ij} \neq w_i$, *otherwise* s_{ij} *is total,* $\mathtt{Var}(s_{ij}) \subseteq \overline{w}$
$-$ *and* $w_i \notin \mathtt{Var}(s_{ij})$ *for every* $w_i \in \overline{w}$ *and* $1 \leq j \leq n_i$.
A formula is in basic normal form *if it is of the form* $Q\overline{y}\ \varphi[\overline{x} \cdot \overline{y}]$ *where* φ *is a disjunction of basic formulas for* $\overline{x} \cdot \overline{y}$. □

As in the case of infinite signatures, the above notion of basic formula is a solved form.

Theorem 4. *Let* Σ *be a finite signature. Any basic formula different from* $false$ *is satisfiable in* $\mathcal{E}^\star$. □

Note that the syntactical form provided in Definition 3 is a particular case of the one in Definition 2. The only difference is that universal quantification is not allowed in the case of finite signatures. Further, there exists a very simple transformation using **E** from formulas as defined in Definition 2 into formulas as defined above.

Proposition 8. *Let* Σ *be a finite signature. Any constraint of the form*

$$\varphi \ = \bigwedge_{w_i \in \overline{w}} \bigwedge_{j=1}^{n_i} (\neg w_i == s_{ij})^{\forall \setminus \overline{w}}$$

can be transformed into an equivalent disjunction of basic formulas for $\overline{w}$. □

Being $\exists \overline{w}\ c(\overline{x}, \overline{w})$ a formula as described in Definition 2, the conjunction of negated equations in $c(\overline{x}, \overline{w})$ is transformed into a disjunction of basic formulas $\overline{w}$ as shown in Proposition 8. Then, the whole formula is transformed into an equivalent disjunction of basic formulas for $\overline{x}$ using **R**, $\mathbf{EE_2}$ and **FT**. This result allows us to easily adapt Propositions 3 and 4 to the case of finite signatures.

Example 6. Let $\mathcal{F}_\Sigma = \{a_{/0}, g_{/1}, f_{/2}\}$ and $\overline{x} = \{x_1, x_2\} \subset \mathcal{V}$. The constraint

$$\exists w\ (\ \neg x_1 == x_1 \wedge x_2 == f(w, a) \wedge \forall v\ \neg w == f(a, v)\ )$$

is transformed into a disjunction of basic formulas for $\overline{x}$ as follows. First, we transform $\forall v\ \neg w == f(a, v)$ into a disjunction of basic formulas for w using **E**:

$$\begin{aligned} &\forall v\ \neg w == f(a, v) \ \wedge \ [\ \neg w == w \ \vee \ w == a \ \vee \ \exists z\ w == g(z) \ \vee \\ &\qquad\qquad \exists \overline{z}\ w == f(z_1, z_2)\ ] \\ \equiv\ &\neg w == w \ \vee \ w == a \ \vee \ \exists z\ w == g(z) \ \vee \\ &\exists \overline{z}\ (\ w == f(z_1, z_2) \wedge \forall v\ \neg w == f(a, v)\ ) \end{aligned} \tag{1}$$

The first three subformulas are already basic formulas for w. Regarding the last one, it is transformed using rules **R** and $\mathbf{D_2}$ as follows

$$\begin{aligned} & \exists \overline{z}\ (\ w == f(z_1, z_2) \wedge [\ \neg z_1 == a\ \vee\ \forall v\ \neg z_2 == v\ ]\ ) \\ \equiv\ & \exists \overline{z}\ (\ w == f(z_1, z_2) \wedge \neg z_1 == a\ )\ \vee \\ & \exists \overline{z}\ (\ w == f(z_1, z_2) \wedge \forall v\ \neg z_2 == v\ ) \end{aligned} \tag{2}$$

where the second subformula is equivalent to $false$ by rules $\mathbf{NFT_3}$ and $\mathbf{NFT_2}$. Thus, $\forall v\ \neg w == f(a, v)$ has been transformed into the disjunction of basic formulas for w in (1, 2). Finally, the conjunction of the above disjunction and $\neg x_1 == x_1 \wedge x_2 == f(w, a)$ is transformed into

$$\begin{aligned} & (\ \neg x_1 == x_1 \wedge x_2 == f(a, a)\ )\ \vee \\ & \exists z\ (\ \neg x_1 == x_1 \wedge x_2 == f(g(z), a)\ )\ \vee \\ & \exists \overline{z}\ (\ \neg x_1 == x_1 \wedge x_2 == f(f(z_1, z_2), a) \wedge \neg z_1 == a\ ) \end{aligned}$$

using rules **R**, $\mathbf{EE_1}$ and $\mathbf{NFT_2}$. □

Finally, in order to be able to apply the algorithm in Figure 4 to the case of finite signatures, we adapt the result in Theorem 3.

Theorem 5. *Let Σ be a finite signature, $\exists \overline{w}\ a(\overline{x} \cdot \overline{y}, \overline{w} \cdot \overline{z})$ a basic formula for the variables $\overline{x} \cdot \overline{y}$ of the form*

$$\exists \overline{w} \cdot \overline{z}\ (\ \bigwedge_{x_1 \in \overline{x}^1} \neg x_1 == x_1 \wedge \bigwedge_{y_1 \in \overline{y}^1} \neg y_1 == y_1 \wedge \overline{x}^2 == \overline{t} \wedge \overline{y}^2 == \overline{r} \wedge \varphi \wedge \psi\ )$$

where $-$ $\overline{w} = \mathtt{Var}(\overline{t})$ *and* $\overline{z} = \mathtt{Var}(\overline{r}) \setminus \overline{w}$,
$-$ φ *is a finite conjunction of negated equations such that* $\mathtt{Var}(\varphi) \subseteq \overline{w}$,
$-$ $\psi = \bigwedge_{i=1}^{n} \neg v_i == s_i$ *and* $(v_i \cup \mathtt{Var}(s_i)) \cap \overline{z} \neq \emptyset$ *for each* $1 \leq i \leq n$.
The formulas $\exists \overline{y}\ [\ \exists \overline{w}\ a(\overline{x} \cdot \overline{y}, \overline{w} \cdot \overline{z})\ ]$ *and* $\exists \overline{w}\ a'(\overline{x}, \overline{w})$ *where*

$$a'(\overline{x}, \overline{w}) \ =\ \bigwedge_{x_1 \in \overline{x}^1} \neg x_1 == x_1 \wedge \overline{x}^2 == \overline{t} \wedge \varphi$$

are equivalent in $\mathcal{E}^\star$. □

5 Conclusions and Future Work

We have axiomatized the theory of infinite trees with strict equality, denoted by either $\mathcal{E}$ (infinite signatures) or $\mathcal{E}^\star$ (finite signatures). Besides, we have provided a decision algorithm, which proves that the theory is complete. Our algorithm follows the proposal in [6] for the equality theory of finite trees. Further, it is easy to see that the problem of deciding first-order equality constraints of finite trees can be reduced to the decision problem of the theory of infinite trees with strict equality: it suffices to restrict the value of every variable x in any formula

to be a finite and total tree by assertions of the form $x == x$. Thus, it follows from the results in [7, 14] that the decision problem of the theory of infinite trees with strict equality is non-elementary (as lower bound).

Although direct applications of our results have been left out of the focus of the paper, we foresee some potential uses that will be subject of future work: Herbrand constraint solvers present in existing functional-logic languages, essentially corresponding to existential constraints, could be enhanced to deal with more general formulas. Constructive failure [10, 9], the natural counterpart of constructive negation in the functional logic field, could also take profit of our methods, specially for the case of programs with extra variables, not considered in the mentioned papers. For these envisaged continuations of our work it could be convenient to extend the theory and methods of this paper by adding two additional predicate symbols: strict disequality (a computable approximation of negation of strict equality) and true equality.

References

1. J. Álvez and F. J. López-Fraguas. A complete axiomatization of strict equality over infinite trees. Technical Report SIC-3-09, UCM, Madrid, 2009.
2. P. Arenas-Sánchez, A. Gil-Luezas, and F. J. López-Fraguas. Combining lazy narrowing with disequality constraints. In M. V. Hermenegildo and J. Penjam, editors, *PLILP'94*, volume 884 of *Lecture Notes in Computer Science*, pages 385–399. Springer-Verlag, 1994.
3. E. J. G. Arias, J. Mariño-Carballo, and J. M. R. Poza. A proposal for disequality constraints in curry. *Electr. Notes Theor. Comput. Sci.*, 177:269–285, 2007.
4. R. Caballero and J. Sánchez (eds.). TOY: A multiparadigm declarative language. Technical report, UCM, Madrid, July 2006.
5. A. Colmerauer. Equations and inequations on finite and infinite trees. In K. L. Clark and S. A. Tärnlund, editors, *FGCS'84*, pages 85–99, 1984.
6. H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3/4):371–425, 1989.
7. K. J. Compton and C. W. Henson. A uniform method for proving lower bounds on the computational complexity of logical theories. *Annals of Pure and Applied Logic*, 48(1):1–79, 1990.
8. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
9. F. J. López-Fraguas and J. Sánchez-Hernández. Failure and equality in functional logic programming. *Electr. Notes Theor. Comput. Sci.*, 86(3), 2003.
10. F. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *Theory and Practice of Logic Programming*, 4(1&2):41–74, 2004.
11. M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *LICS'1988*, pages 348–357. IEEE Computer Society, 1988.
12. S. e. Peyton Jones. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge Univ. Press, 2003.
13. R. Reiter. On closed world data bases. *Logic and Data Bases*, pages 55–76, 1978.
14. S. G. Vorobyov. An improved lower bound for the elementary theories of trees. In M. A. McRobbie and J. K. Slaney, editors, *CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 275–287. Springer-Verlag, 1996.

Integrating ILOG CP technology into $\mathcal{TOY}^\star$

Nacho 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 (FD) constraint solver. In this work, we show how to integrate the ILOG CP 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 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 [10]. 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 H), linear and non-linear arithmetic constraints over reals (R), finite domain constraints over integers (FD), and a communication domain (M) which makes possible the cooperation among H, R and FD. Whereas R as FD rely on the constraint solvers provided by SICStus Prolog, solving in H and M needs an explicitly management [3]. $\mathcal{TOY}$ offers a wide range of finite domain constraints comparable to many CLP(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 FD constraint system is shown to the left of Fig. 1. $\mathcal{TOY}$ identifies each 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 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: $\mathcal{TOY}(FDs)$

$\mathcal{TOY}$ (referred to as $\mathcal{TOY}(FDs)$ from now on) has been using the 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 FD constraint system is shown in the middle of Fig. 1. Next, we show a basic example for illustrating the use of the system $\mathcal{TOY}(FDs)$ with finite domains constraints.

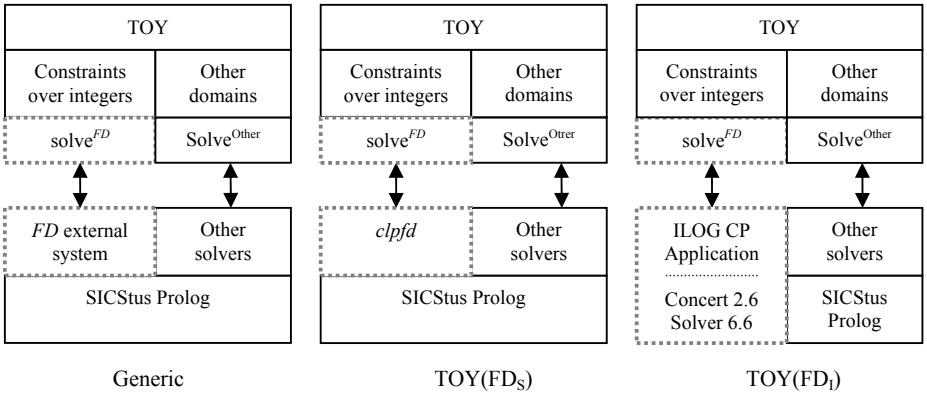


Fig. 1. Architectural Components

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 $\mathcal{TOY}(FDs)$ 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 FD system has some disadvantages:

- Recent works [2] have proved that its performance can be enhanced, 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 [6] is an industrial technology market leader. Its nature is declarative and provides a C++ API to access its libraries. Its 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 constraints set input. Its wide range of global constraints make possible to formulate different and complex properties. The use of different constraint solvers for a unique application domain is also allowed. Moreover, libraries for solving specific, efficient algorithms for 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 [7]. 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 FD constraint solving:

- ILOG Solver 6.6, for generic 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.

As a first approach, we will consider only ILOG Solver 6.6. For this case, any ILOG CP application needs the following set of ILOG Concert 2.6 and ILOG Solver 6.6 objects (see [6] for a detailed explanation):

- `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 FD problem, which are:
  - `IloIntVarArray` ***vars***`(env)` This vector is intended to make possible to reference all of the decision variables of the model from a unique object. Each variable must be created previously by `IloIntVar` ***v***`(env, int lowerBound, int upperBound)`.
  - `IloConstraint` ***c*** Each `IloConstraint` involves some `IloIntVar` of `vars`. It can be added directly to the model, without being created previously.
- `IloSolver` ***solver***`(env)` It is the main solving object. It contains an object `IloGoal` ***goal*** which specifies the concrete search procedure to be used. ***solver*** main methods are:
  - `solver.extract(model)` Extracts the information contained into `model`. For each `IloIntVar` and `IloConstraint` contained in `model` it creates an associated new `IlcIntVar` or `IlcConstraint` object.
  - `solver.solve(goal)` Solves the extracted model.

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

In this section, we explain in detail how to integrate ILOG CP FD technology into the system $\mathcal{TOY}$ (referred to as $\mathcal{TOY}(FDi)$ 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 $\mathcal{TOY}(FDi)$ is shown.

2.1 Connecting SICStus Prolog with C++

It is possible to communicate a SICStus Prolog application with a C++ component. This communication is done by mapping a set of linking Prolog facts (contained in the Prolog application) with a set of C++ functions (defined in the C++ component). The C++ component needs to be a dynamic library with a specific internal file structure. SICStus Prolog also defines a set of possible conversions between Prolog arguments and C++ arguments. Each arguments of a linking Prolog fact 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 fact 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 facts 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 needs two files for creating a dynamic library as, for instance `interface.dll`, which could be used within a SICStus Prolog application:

- `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){...}
  ```

SICStus Prolog supplies a tool, `splfr` [9], to create a dynamic library (say `interface.dll`), taking as input `interface.pl` and `interface.cpp`. The macro `splfr` is used as a shortcut to the execution of some compiling and linking commands offered by Microsoft Visual C++ [8]. 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 implement $\mathcal{TOY}(FDi)$ in such a way it accepts any $\mathcal{TOY}(FDs)$ input goal, including all FD constraints managed by the existing $solve^{FD}$ in $\mathcal{TOY}(FDs)$. Also, $\mathcal{TOY}(FDi)$ uses the same goal solution structure as $\mathcal{TOY}(FDs)$ does. To achieve that behavior is necessary to solve the following difficulties:

- As $\mathcal{TOY}$ is a system implemented in SICStus Prolog, in $\mathcal{TOY}(FDs)$ the communication between $\mathcal{TOY}$ and its FD technology is quite natural. However, as ILOG CP is implemented in C++, some glue code is needed to fix the impedance mismatch problem.
- ILOG CP and SICStus Prolog differ on their notion of solution of a FD problem.

There have been four difficult tasks to achieve in the new system $\mathcal{TOY}(FDi)$. 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 FD solver, and we use `vars` if we refer to the decision variables contained in `model`.

Managing Decision Variables

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

– We need to create as many `IloIntVar` decision variables as FD logic variables take part into the FD constraint set. These variables must be added to `model` and `vars` (the former to model the FD problem properly and the latter to make possible to refer to each variable of the model from a unique object).
– We must find a bijective relation that associates each FD logic variable of the $\mathcal{TOY}$ goal with each decision variable existing in the ILOG CP vector `vars`.
– We model each FD constraint in ILOG CP over the set of decision variables of the vector `vars` associated to the set of FD logic variables involved in that FD constraint.

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

- The FD logic variable contained in $\mathcal{TOY}$.
- The decision variable modeled as an `IloIntVar` object in `model`.
- The specific `IlcIntVar` object created by `solver` from its associated `IloIntVar` object contained in `model`.

A first attempt for mapping a 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` conversion of the FD logic variables. Each time $solve^{FD}$ sends a new FD constraint to ILOG CP, the associated C++ function will first look for its FD logic variables into the `SP_term_ref` vector. If it can not find any variable, we can assure that the C++ function is dealing with a new FD logic variable not treated before. So, the C++ function adds this new 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 FD logic variable of the FD constraint sent by $solve^{FD}$ is contained at an index of the `SP_term_ref` vector, the 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_ref`s (as arguments or dynamically created within it) finishes its execution, all these `SP_term_ref`s become invalid. Let's see the next example, where an interface between the Prolog predicates `p1`, `p2` and `p3` and the C++ functions `f1`, `f2` and `f3`, resp, is defined. Functions `f1` and `f2` receive a Prolog term as an argument, while `f3` receives two Prolog terms.

– Let's call `p3` with to occurrences of the logic variable `X`, as `p3(X,X)`. If we make `SP_compare(t1,t2)` within `f3(SP_term_ref t1, SP_term_ref t2)` the result says that both `SP_term_ref`s are in fact the same Prolog term.
– But, let's do the call `p1(X)`. We store `t1` of `f1(SP_term_ref t1)` into a global vector <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 make `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` finish, 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 into the Prolog application by the use of a list of FD logic variables (referred to as `L` from now on). We want `L` to be used in each $solve^{FD}$ predicate. On one hand SICStus Prolog does not allow global variables. On the other hand, there is a logic variable `Cin` [4], which represents a mixed constraints 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 `L`, into `Cin`. Each time a $solve^{FD}$ predicate manages a new FD constraint, we can check whether a FD logic variable belongs to `L` or not by accessing to it within `Cin`. Any new FD logic variable is automatically added to the end of `L`, 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 FD logic variables of the FD constraint belongs to `L`, $solve^{FD}$ determines their indexes, and put them as arguments to the C++ function, which models the 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 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 terms unification which results from the Herbrand equality constraint `X == 3`. This unification is visible at any occurrence of that FD logic variable, particularly the one in `L`. This causes an inconsistency between the contents of `L` 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 FD logic variable. Thus, each time the Herbrand solver binds a 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 FD logic variable in `L`. But, unfortunately, the arguments of `verify_attributes(+Var, +Value, +Goals)` are fixed. SICStus Prolog does not allow global variables, so there is no way to get access to `L`.

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 `L`. Now it becomes a list of pairs. The first element of each pair contains the FD logic variable, and the second one contains a flag which determines if the bound FD logic variable has been synchronized with `vars`. Thus, while the FD logic variable is not bound, the value of the flag remains at `0`. When the 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 FD constraint to ILOG CP, it must previously:

- Look for any pair in `L` (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 `L` by `[`*`value`*`,1]`

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

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

The first FD logic variable of the goal is `X`, which occurs at the first position of `L` and `vars`. The synchronization of `X == 1` as `vars[0] == 1` makes the FD problem infeasible. So, the $\mathcal{TOY}$ goal will fail after `X == 1`, and there is no need of computing the rest of the goal expressions. However, the first equality `vars[0] == 1` is not computed until the next FD constraint is posted.
As `X == 1, X1 == 1, X2 == 1, ... , X1000 == 1` are computed by Herbrand solver there are no more 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 set of C++ functions concerning the management of FD constraints. This produces a lack of synchronization between the vector `vars` and `L`. To achieve the synchronization, whenever any of this C++ functions binds to *value* `vars[i]`, the pair contained at position `i` of `L` must be automatically unified with `[`*`value`*`,1]`.

To this end, $solve^{FD}$ sends `L` to a C++ function as an input argument, and puts an output argument to obtain the new state of `L` computed within the C++ function. A new global variable of type vector`<int,int>` must be created

in ILOG CP. This vector of pairs is cleared at the beginning of each C++ function. Each pair of the vector contains:

- The index `i` in `vars` of the decision variable.
- The *value* that `solver` has obtained for this variable.

A C++ function manages any new constraint by adding it to `model`, and propagates its new FD constraint set. Next, the C++ function accesses to the contents of the vector`<int,int>`, to see whether there are any `IloIntVar` that has been bound. Using the content of vector`<int,int>` and `L`, the C++ function builds the new state of `L` by unifying as many FD logic variables as vector`<int,int>` demands.

The only remaining task to be explained is how to add each pair to the global 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 its `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 bounded `IloIntVar` 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` and `model`, we impose the constraint `IlcCheckWhenBound`.
- When this `IloIntVar` becomes bound, `IlcCheckWhenBound` propagates, triggering the demon `RealizeVarBound`.
- `RealizeVarBound` executes the `IlcCheckWhenBound` method `varDemon`, which adds the pair `<index of the variable, value of the variable>` to vector`<int,int>`.

Solutions in ILOG CP

Any $\mathcal{TOY}(FDs)$ solution is expressed in general with constraints (equality, disequality, FD constraints –including ranges–). Of course, $\mathcal{TOY}(FDs)$ accepts to label FD variables by calling the FD labeling enumeration procedure.

In $\mathcal{TOY}(FDi)$, to show the remaining values of the FD logic variables we access to each `IlcIntVar` of `solver` by its associated `IloIntVar` contained in `model`. There are some methods to check the remaining values of these variables. However, ILOG Solver does not grant access to simplified constraints (i.e., solved forms). The ILOG philosophy of a solution is to select a value for each decision variable while satisfying the constraint set. Of course, you can use no search procedure, obtaining the same structure as in an interval solution, but again without accessing the simplified constraints. As in our context we have to show them, we store within `Cin` a list with the FD constraints (referred to as `C` from now on) appearing in the $\mathcal{TOY}$ goal.

2.3 A $\mathcal{TOY}(FDi)$ Example

In this section we detail how goal solving works with the new system $\mathcal{TOY}(FDi)$ over the 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 expression evaluation. On one hand we look at the state of `L` and `C` within `Cin`. On the other hand we look at the state of `vars`, `model`, `solver` by pointing out any `IloIntVar`, `IloConstraint`, `IlcIntVar`, `IlcConstraint` object accessed through them. For each goal expression any new element added to each data structure is remarked in boldface. Figure 2 tries to make it clearer:

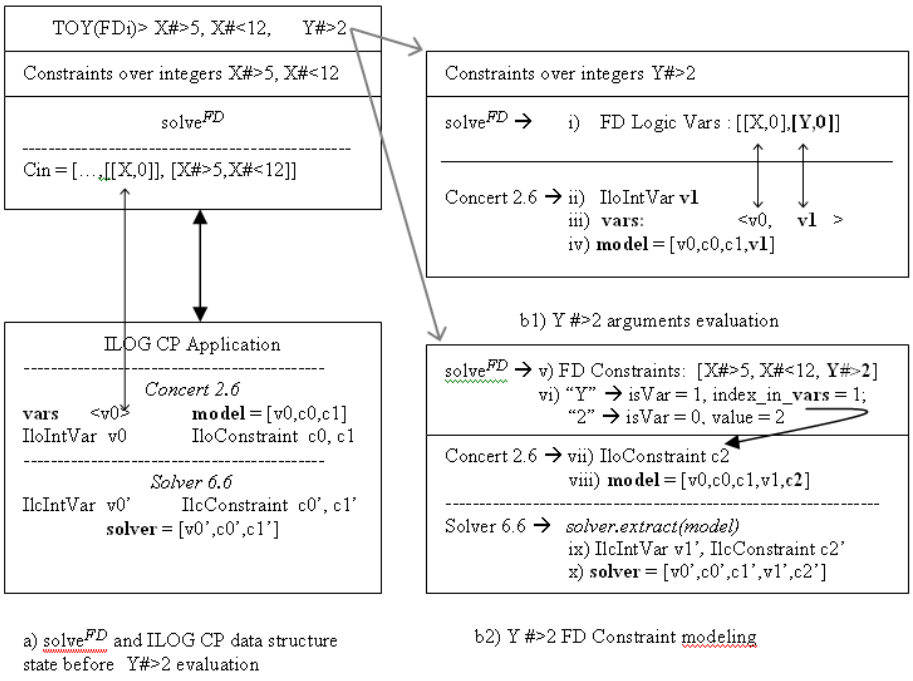


Fig. 2. $TOY(FDi)$ data structures evolution over FD Constraint expression evaluation

- Figure 2a) represents the internal state of $solve^{FD}$ and ILOG CP data structures at the end of `Toy(FDi)> X #>= 5, X #<= 12` evaluation.
- Figure 2b1) and 2b2) describes which actions must be done for the correct management of the new FD Constraint `Y #>= 2`.

Before evaluating any goal expression, in the *solve*^{FD} side `L`=[] and `C`=[]. In the ILOG CP side `model`=[], `vars`=`<>` and `solver`=[]. There is also no `IloIntVar`, `IloConstraint`, `IlcIntVar`, `IlcConstraint` objects.

- Execution of `X #>= 5`
  The new *FD* constraint is added to `C`=[**X#>=5**]. The new *FD* logic var is added to `L`=[[**X,0**]]. A new `IloIntVar` **v0** is created and added to `vars`=**<v0>** and `model`=[**v0**]. A new `IloConstraint` **c0** is created, involving `vars[0]` and the value 5. This `IloConstraint` **c0** is added to `model`=[v0,**c0**]. `solver` extracts the new state of `model` and creates a new `IlcIntVar` **v0'** and a new `IlcConstraint` **c0'**. `solver`=[**v0'**,**c0'**]. Its constraint propagation technique prunes the domain of **v0'**=5..sup. The state of the solver remains 'Feasible'. $\mathcal{TOY}$ continues evaluating next goal expression.
- Execution of `X #<= 12`
  `C`=[`X#>=5`,**X#<=12**]. `L`=[`[X,0]`]. `vars`=`<v0>`. A new `IloConstraint` **c1** is created involving `vars[0]` and 12. `model`=[v0,c0,**c1**]. `solver` extracts `model` creating `IlcConstraint` **c1'**. `solver`=[v0',c0',**c1'**]. Constraint propagation prunes **v0'**=5..12. `solver` state='Feasible'.
- Execution of `Y #>= 2`
  By managing `Y#>=2` arguments, *solve*^{FD} adds `L`=[`[X,0]`,[**Y,0**]]. By adding a new *FD* Logic Var to `L`, a new `IloIntVar` **v1** is created and added to `vars`=`<v0,v1>` and `model`[v0,c0,c1,**v1**]. There is a correspondence between **Y** and **v1** because both are at the same position of `L` and `vars` respectively. *solve*^{FD} adds `Y#>=2` to `C`=[`X#>=5,X#<=12`,**Y#>=2**]. The relevant information to modeling the *FD* constraint into ILOG CP is the tuple `<1,1,0,2>` which says if the arguments are variables or not and its index/value respectively. Then a new `IloConstraint` **c2** is created involving `vars[1]` and the value 2. This `IloConstraint c2` is added to `model`=[v0,c0,c1,v2,**c2**]. `solver` extracts the new state of `model` creating a new `IlcIntVar` **v1'** and `IlcConstraint` **c2'**. `solver`=[v0',c0',c1',**v1'**, **c2'**]. Constraint propagation prunes **v1'**=2..sup. `solver` state='Feasible'. After constraint propagation, the program control comes back to *solve*^{FD}. It finishes the management of the *FD* constraint by storing the new state of `L`=[`[X,0],[Y,0]`] and `C`=[`X#>=5,X#<=12,Y#>=2`] into `Cin`.
- Execution of `Y #<= 17`
  `C`=[`X#>=5,X#<=12,Y#>=2`,**Y#<=17**]. `L`=[`[X,0],[Y,0]`]. `vars`=`<v0,v1>`. A new `IloConstraint` **c3** is created involving `vars[1]` and 12.
  `model`=[v0,c0,c1,v2,c2,**c3**]. `solver` extracts `model` creating `IlcConstraint` **c3'**. `solver`=[v0',c0',c1',v1',c2',**c3'**]. Constraint propagation prunes **v1'**=2..17. `solver` state='Feasible'.
- Execution of `X#+Y==17`
  This expression includes a compound constraint. This constraint must be decomposed into primitive constraints. In this case: `X#+Y==_Z, _Z==17`
  - Execution of `X#+Y==_Z`
    `C`=[`X#>=5,X#<=12,Y#>=2,Y#<=17`,**X#+Y==_Z**].
    `L`=[`[X,0],[Y,0],`[**_Z,0**]]. A new `IloIntVar` **v0** is created and added

to vars=<v0,v1,**v2**>. A new IloConstraint **c4** is created involving vars[0] and vars[1]. model=[v0,c0,c1,v2,c2,c3,**c4**]. solver extracts model creating IlcIntVar **v2'** and IlcConstraint **c4'**.
solver=[v0',c0',c1',v1',c2',c3',**v2'**,**c4'**]. Constraint propagation prunes **v2'**=7..29. solver state='Feasible'.

- Execution of _Z==17
  $\mathcal{TOY}$ sends _Z == 17 to the Herbrand solver. This will bind the variable _Z to 17, L=[[X,0],[Y,0],[**17,0**]],
  C=[X#>=5,X#<=12,Y#>=2,Y#<=17,**X#+Y==17**]. However, this value will not be automatically synchronized with ILOG CP. The synchronization will happen before either a new FD constraint is sent or at the end of the $\mathcal{TOY}$ goal.

– Execution of X#-Y==5
This expression is decomposed again into X#-Y==_T, _T==5

- Execution of X#-Y==_T
  As we have pointed out, before the new FD constraint is sent to ILOG CP, any pattern [*value*,0] contained in L at position i will be synchronized with model by adding the new IloConstraint vars[i]==*value*.
  C=[X#>=5,X#<=12,Y#>=2,Y#<=17,X#+Y==17].
  L=[[X,0],[Y,0],[17,0]]. A new IloConstraint **c5** is created involving vars[2] and 17.
  model=[v0,c0,c1,v2,c2,c3,c4,**c5**]. solver extracts model creating IlcConstraint **c5'**. solver=[v0',c0',c1',v1',c2',c3',v2',c4',**c5'**].
  Constraint propagation bounds vars[2] to 17. L=[[X,0],[Y,0],[**17,1**]].
  solver state='Feasible'.

  As there is no more patterns [*value*,0] in L, $solve^{FD}$ is now able to manage the constraint X#-Y==_T. So the new FD constraint is added to
  C=[X#>=5,X#<=12,Y#>=2,Y#<=17,X#+Y==17,**X#-Y==_T**].
  L=[[X,0],[Y,0],[17,1],[**_T,0**]]. A new IloIntVar **v0** is created and added to vars=<v0,v1,v2,**v3**>. model=[v0,c0,c1,v2,c2,c3,c4,c5,**v3**]. A new IloConstraint **c6** is created involving vars[0] and vars[1].
  model=[v0,c0,c1,v2,c2,c3,c4,c5,v3,**c6**].solver extracts model creating IlcIntVar **v3'** and IlcConstraint **c6'**.
  solver=[v0',c0',c1',v1',c2',c3',v2',c4',c5',**v3'**,**c6'**].
  Constraint propagation prunes **v0'**=6..12, **v1'**=5..11', **v3'**=1..7.
  solver state='Feasible'.
- Execution of _T==5
  $\mathcal{TOY}$ sends _T == 5 to the Herbrand solver. This will bind the variable _T to 5, making L=[[X,0],[Y,0],[17,1],[**5,0**]],
  C=[X#>=5,X#<=12,Y#>=2,Y#<=17,X#+Y==17,**X#-Y==5**].
  Again, the synchronization will happen before either a new FD constraint is sent or at the end of the $\mathcal{TOY}$ goal.

– The $\mathcal{TOY}$ goal is almost finished. To completely finish the goal computation we synchronize the pairs L with the pattern [*value*,0].
C=[X#>=5,X#<=12,Y#>=2,Y#<=17,X#+Y==17].

L=[[X,0],[Y,0],[17,1],[5,0]]. A new IloConstraint **c7** is created involving vars[3] and 5. model=[v0,c0,c1,v2,c2,c3,c4,c5,v3,c6,**c7**]. solver extracts model creating IlcConstraint **c7'**.
solver=[v0',c0',c1',v1',c2',c3',v2',c4',c5',v3',c6',**c7'**]. Constraint propagation bounds vars[3] to 5, **v0'**=10..12, **v1'**=5..7'. L=[[X,0],[Y,0],[17,1],[**5,1**]]. solver state='Feasible'.
After this synchronization, the $\mathcal{TOY}$ goal is completely finished. It shows as the computed answer the set of non-ground FD constraints of C as well as the (unbound) variables of L. For each of these variables, $\mathcal{TOY}$ shows its domain. These values are obtained from the IlcIntVar contained in solver through the associated IloIntVar contained in model. Each decision variable of model is accessed through its position of vars.

```
      yes
      { X #+ Y #= 17,
        X #- Y #= 5,
        X in 10..12,
        Y in 5..7 }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

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 $\mathcal{TOY}(FDs)$ and $\mathcal{TOY}(FDi)$ 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 two labeling strategies: a static search procedure which selects the variables in the textual order they occur in the program, and the dynamic search procedure 'first fail' (denoted by ff), which selects first the variable with minimum domain size. For a given variable, both of them selects first the minimum value in its domain.

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

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

The benchmarks programs are:

- 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, $\mathcal{TOY}(FDi)$ takes more time than $\mathcal{TOY}(FDs)$ 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 $\mathcal{TOY}(FDi)$ is slower than $\mathcal{TOY}(FDs)$. This is because of the time involved in the communication between the Prolog implementation of $\mathcal{TOY}(FDi)$ and ILOG CP. However, as the nodes needed to be explored increase slightly, this waste of time is balanced, making $\mathcal{TOY}(FDi)$ more efficient than $\mathcal{TOY}(FDs)$.
- The solution `[X1,_,Xn]` holds: $\forall i \in \{1..n\}$ `Xi` $= n - (i - 1)$. 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

The above conclusions are clearly confirmed in this second benchmark, where $\mathcal{TOY}(FDi)$ is faster than $\mathcal{TOY}(FDs)$ 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, $\mathcal{TOY}(FDi)$ takes

slightly less solving time than $\mathcal{TOY}(FDs)$. In any case, these measurements point out that our first approach to integrate the ILOG CP technology into $\mathcal{TOY}(FDi)$ is encouraging, but also that the management of the additional data structures used for the interface should be optimized.

4 Conclusions and Future Work

In this work, we have studied how to integrate the FD ILOG CP technology into the system $\mathcal{TOY}$. We have shown that this technology offers some advantages over the existing system $\mathcal{TOY}$ based on the 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 $\mathcal{TOY}(FDi)$ is faster than $\mathcal{TOY}(FDs)$ 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 FD component. Therefore, optimizing this management will be the target of our immediate future work. In addition, backtracking management will be covered in a next work, together with an extended set of benchmarks. Another subject of interest is to test other constraint libraries, as Gecode [11].

References

1. P. Arenas, S. Estévez, A. Fernández, A. Gil, F. López-Fraguas, M. Rodríguez-Artalejo, and F. Sáenz-Pérez. $\mathcal{TOY}$. a multiparadigm declarative language. version 2.3.1., 2007. R. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`.
2. R. G. del Campo and F. Sáenz-Pérez. Programmed Search in a Timetabling Problem over Finite Domains. *Electr. Notes Theor. Comput. Sci.*, 177:253–267, 2007.
3. S. Estévez-Martín, A. Fernández, M. Hortalá-González, F. Sáenz-Pérez, M. Rodríguez-Artalejo, and R. del Vado-Vírseda. On the Cooperation of the Constraint Domains H, R and FD in $CFLP$. *Theory and Practice of Logic Programming*, 2009. Accepted for publication.
4. S. Estévez-Martín, A. J. Fernández, and F. Sáenz-Pérez. About implementing a constraint functional logic programming system with solver cooperation. In *proc. of CICLOPS'07*, pages 57–71, 2007.
5. A. J. Fernández, T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. *Theory Pract. Log. Program.*, 7(5):537–582, 2007.
6. ILOG. ILOG Solver 6.6, Reference Manual, 2008.
7. ILOG. ILOG OPL Studio 6.1, Reference Manual, 2009.
8. Microsoft, 2005. `http://msdn.microsoft.com/en-us/visualc/default.aspx`.
9. SICStus Prolog. Using SICStus Prolog with newer Microsoft C compilers.
10. SICStus Prolog, 2007. `http://www.sics.se/isl/sicstus`.
11. The Gecode team. Generic constraint development environment. Available from http://www.gecode.org, 2006.